Functions

When we call a function:

- the arguments are evaluated and set up for function
- control is transferred to the code for the function
- local variables are created
- the function code is executed in this environment
- the return value is set up
- control transfers back to where the function was called from
- the caller receives the return value
Functions

Data associated with function calls is placed on the MIPS stack.

![Diagram showing the MIPS stack with sections for stack frames, program data, program code, and reserved areas for the kernel.](image)
Functions

Each function allocates a small section of the stack (a frame)
- used for: saved registers, local variables, parameters to callees
- created in the function prologue (pushed)
- removed in the function epilogue (popped)

Why we use a stack:
- function $f()$ calls $g()$ which calls $h()$
- $h()$ runs, then finishes and returns to $g()$
- $g()$ continues, then finishes and returns to $f()$

i.e. last-called, exits-first (last-in, first-out) behaviour
Functions

How stack changes as functions are called and return:
Register usage conventions when \( f() \) calls \( g() \):

- **caller saved registers (saved by \( f() \))**
  - \( f() \) tells \( g() \) "If there is anything I want to preserve in these registers, I have already saved it before calling you"
  - \( g() \) tells \( f() \) "Don’t assume that these registers will be unchanged when I return to you"
  - e.g. \$t0 .. \$t9, \$a0 .. \$a3, \$ra

- **callee saved registers (saved by \( g() \))**
  - \( f() \) tells \( g() \) "I assume the values of these registers will be unchanged when you return"
  - \( g() \) tells \( f() \) "If I need to use these registers, I will save them first and restore them before returning"
  - e.g. \$s0 .. \$s7, \$sp, \$fp
Aside: Branch Delay Slots

Most architectures are “pipelined” to improve efficiency.
  - next instruction(s) can start before previous one(s) finish
  - MIPS has a two-stage pipeline: fetch, and execute

What if a branching instruction (e.g. `jal`) is executed?
  - next fetch happens before branch executes $\Rightarrow$ next instruction is loaded.
  - so, either ‘stall’ the pipeline until it completes (slow!)
  - $\ldots$ or run the next instruction anyway, in a branch-delay slot

On classical MIPS, the next instruction is executed **before** the branch completes.
Aside: Branch Delay Slots

To avoid potential problems, use nop immediately after branch
A problem scenario, and its solution (branch delay slot):

```
# Implementation of print(compute(42))
li $a0, 42
jal compute
move $a0, $v0
jal print
```

Since SPIM is not pipelined (by default), the nop is not required
Aside: Why do we need both $fp and $sp?

During execution of a function

- $sp can change (e.g. pushing params, adding local vars)
- may need to reference local vars on the stack
- useful if they can be defined by an offset relative to fixed point
- $fp provides a fixed point during function code execution

```c
int f(int x) {
    int y = 0;  // y created in prologue
    for (int i = 0; i < x; i++) {
        y += i;  // i created in for-loop
            // which changes $sp
    }
    return y;
}
```

It is possible to only use $sp, which frees up $fp to become $s8. But at the cost of lots of extra work for us in keeping track of stack layout.
Simple view of function calls:

- load argument values into $a0, $a1, ...
- invoke jal: loads PC into $ra, jumps to function
- function puts return value in $v0
- returns to caller using jr $ra
Function Calling Protocol

Before one function calls another, it needs to

- place (32-bit) arguments in the $a0..$a3
- if more than 4 args, or args larger than 32-bits ...
  - push value of all such args onto stack
- save any non-$s$? registers that need to be preserved
  - push value of all such registers onto stack
- jal address of function (usually given by a label)

Pushing value of e.g. $t0$ onto stack means:

```
addi $sp, $sp, -4 OR sw $t0, -4($sp)
sw $t0, ($sp)
addi $sp, $sp, -4
```
**Example: simple function call**

```c
int main(void) {
    // x is $s0, y is $s1, z is $s2
    int x = 5;
    int y = 7;
    int z;
    ...
    z = sum(x, y, 30);
    ...
}

int sum(int a, int b, int c) {
    return a + b + c;
}
```
Function Calling Protocol

Simple function call:

Memory

main()'s stack frame

sum()'s stack frame will go here

Registers

<table>
<thead>
<tr>
<th>argument a from x ($s0)</th>
<th>argument b from y ($s1)</th>
<th>argument c from const</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a0</td>
<td>$a1</td>
<td>$a2</td>
</tr>
</tbody>
</table>

```
# ...  
# z = sum(x, y, 30);  
# ...  
# set up args  
move $a0, $s0  
move $a1, $s1  
li $a2, 30  
# call sum()  
jal sum  
# $s2 = return value  
moves $s2, $v0
```
Function Calling Protocol

Execution of `sum()` function:

```
sum:
  # set up stack frame
  move $v0, $a0
  add  $v0, $v0, $a1
  add  $v0, $v0, $a2
  jr   $ra
```

Memory

- `main()`'s stack frame
- `sum()`'s stack frame

Registers

- `argument a from x ($s0)` - $a0
- `argument b from y ($s1)` - $a1
- `argument c from const` - $a2
Write MIPS code to implement the function call and the function body in ...

char a[100];
int main(void) {
    fgets(a, 99, stdin);
    printf("%d\n", length(a, 99));
    return 0;
}

int length(char *s, int n) {
    int nchars = 0;
    char *end = &s[n];
    while (s < end && *s != \0) {
        s++;
        nchars++;
    }
    return nchars;
}
Functions in MIPS have the following general structure:

# start of function

FuncName:
   # function prologue
   # set up stack frame ($sp, $fp if used)
   # save relevant registers (e.g. $ra)
   ...

   # function body
   # perform computation using $a0, etc.
   # leaving result in $v0
   ...

   # function epilogue
   # restore saved registers (e.g. $ra)
   # clean up stack frame ($sp, $fp if used)
   jr  $ra

Aim of prologue: create environment for function to execute in.
Contents of a typical stack frame:

- stack frame of caller
  - value of previous $fp
  - saved value of $ra
  - space for saved $s? registers
  - space for non-register local variables
  - large arguments for called functions are placed here (changes $sp)

- Higher addresses
  - $fp

- Lower addresses
  - $sp
Before a function starts working, it needs to ...  
- create a stack frame for itself (change $fp$ and $sp$)  
- save the return address ($ra$) in the stack frame  
- save any $s$? registers that it plans to change  

We can determine the initial size of the stack frame via  
- 4 bytes for saved $fp$ + 4 bytes for saved $ra$  
- + 4 bytes for each saved $s$?  

Changing $fp$ and $sp$ ...  
- new $fp = old \ sp - 4$  
- new $sp = old \ sp - size$ of frame (in bytes)
Function Prologue

Example of function `fx()`, which uses `$s0`, `$s1`, `$s2`

```assembly
# start of fx() function
fx:
# fx()’s prologue
sw $fp, -4($sp)
sw $ra, -8($sp)
sw $s0, -12($sp)
sw $s1, -16($sp)
sw $s2, -20($sp)
la $fp, -4($sp)
add $sp, $sp, -20
# rest of fx()’s code
```

Initial values: `$fp`, `$sp`
Final values: `$fp`, `$sp`
Function Prologue

Alternatively ... (more explicit push)

```
# start of fx() function
fx:
# fx()'s prologue
addi $sp, $sp, -4
sw $fp, ($sp)
move $fp, $sp
addi $sp, $sp, -4
sw $ra, ($sp)
addi $sp, $sp, -4
sw $s0, ($sp)
addi $sp, $sp, -4
sw $s1, ($sp)
addi $sp, $sp, -4
sw $s2, ($sp)
# rest of fx()'s code
initial values: $fp, $sp
final values: $fp, $sp
```
Function Prologue

Alternatively ... (relative to new \$fp)

```
# start of fx() function
fx:
  # fx()'s prologue
  sw $fp, -4($sp)
  la $fp, -4($sp)
  sw $ra, -4($fp)
  sw $s0, -8($fp)
  sw $s1, -12($fp)
  sw $s2, -16($fp)
  add $sp, $sp, -20
  # rest of fx()'s code

initial values: $fp,$sp
final values: $fp,$sp
```
Before a function returns, it needs to ...

- place the return value in $v0 (and maybe $v1)
- pop any pushed arguments off the stack
- restore the values of any saved $s? registers
- restore the saved value of $ra (return address)
- remove its stack frame (change $sp and $fp if used)
- return to the calling function (jr $ra)

Locations of saved values computed relative to $fp

Changing $fp and $sp ...

- new $sp = old $fp + 4, new $fp = memory[old $fp]
Example of function `fx()`, which uses `$s0`, `$s1`, `$s2`
Exercise: Function to compute $1+2+3+\ldots+n$

Implement the function `sumTo()`

```c
int main(void) {
    int max;
    printf("Enter +ve integer: ");
    scanf("%d", &max);
    printf("Sum 1..%d = %d\n", max, sum_to(max));
    return 0;
}

int sum_to(int n) {
    int sum = 0;
    for (int i = 1; i <= n; i++) {
        sum += i;
    }
    return sum;
}
```