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
Function Calls

Simple view of function calls:

- load argument values into $a0..$a1
- invoke : loads PC+4 into $ra, jumps to function
- function puts return value in $v0
- returns to caller using $ra
Function with No Parameters or Return Value

- jal hello sets $ra to address of following instruction and transfers execution to hello
- jr ra ∗ ∗transfersexecutiontotheaddressin ∗ ∗ra

```c
int main(void) {
    hello();
    return 0;
}

void hello(void) {
    printf("hi\n");
}
```

```
main:
...                          main:
    jal hello                  ...  
    ...                        

hello:
    la $a0, string
    li $v0, 4
    syscall
    jr $ra
.data
string:
    .asciiz "hi\n"
```
Function with a Return Value but No Parameters

- by convention return value is passed back in $v0

```c
int main(void) {
    int a = answer();
    printf("%d\n", a);
    return 0;
}

int answer(void) {
    return 42;
}
```

```
main:
    ...  
jal    answer
    move $a0, $v0
    li     $v0, 1
    syscall

answer:
    ...  
    li     $v0, 42
    jr     $ra
```
Function with a Return Value and Parameters

- by convention first 4 parameters passed in $a0 .. $a3
- if there are more parameters they are passed on the stack

```c
int main(void) {
    int a = product(6, 7);
    printf("%d\n", a);
    return 0;
}

int product(int x, int y) {
    return x * y;
}
```

```asm
main:
    ...;
    li $a0, 6
    li $a1, 7
    jal product
    move $a0, $v0
    li $v0, 1
    syscall
    ...;

product:
    mul $v0, $a0, $a1
    jr $ra
```
Function calling another function - How NOT to Do It

- a function that calls another function must save $ra
- in the example below jr

  $ra \star \star \text{inmainwillfailbecause} \star \star \text{jalhello} \star \star \text{changed} \star \star \text{$ra}$

```c
int main(void) {
    hello();
    return 0;
}

void hello(void) {
    printf("hi\n");
}
```

```
main:
    jal  hello
    li   $v0, 0
    # THIS WILL FAIL
    jr   $ra

hello:
    la   $a0, string
    li   $v0, 4
    syscall
    jr   $ra

.data
string: .asciiz "hi\n"
```
Stack - Where it is in Memory

Data associated with a function call placed on the stack:
$sp$ (stack pointer) initialized by operating system
- always 4-byte aligned (divisible by 4)
- points at currently used (4-byte) word
- grows downward
- a function can do this to allocate 40 bytes:

```assembly
sub $sp, $sp, 40  # move stack pointer down
```

- a function **must** leave $sp$ at original value
- so if you allocated 40 bytes, before return ($jr \$ra$)

```assembly
add $sp, $sp, 40  # move stack pointer back
```
Stack - Using stack to Save/Restore registers

f:

```
sub $sp, $sp, 12  # allocate 12 bytes
sw $ra, 8($sp)  # save $ra on $stack
sw $s1, 4($sp)  # save $s1 on $stack
sw $s0, 0($sp)  # save $s0 on $stack

... 

lw $s0, 0($sp)  # restore $s0 from $stack
lw $s1, 4($sp)  # restore $s1 from $stack
lw $ra, 8($sp)  # restore $ra from $stack
add $sp, $sp, 12  # move stack pointer back
jr $ra  # return
```
How stack changes as functions are called and return:
Function calling another function - How to Do It

- a function that calls another function must save $ra

```assembly
main:
  sub $sp, $sp, 4  # move stack pointer down
                     # to allocate 4 bytes
  sw $ra, 0($sp)  # save $ra on $stack

jal hello  # call hello

lw $ra, 0($sp)  # recover $ra from $stack
add $sp, $sp, 4  # move stack pointer back up
                     # to what it was when main called
li $v0, 0  # return 0
jr $ra  #
```
MIPS Register usage conventions

- $a0..a3$ contain first 4 arguments
- $v0$ contains return value
- $ra$ contains return address
- if function changes $sp * *, **fp, s0..s7$ it restores their value
- callers assume $sp * *, **fp, s0..s7$ unchanged by call (jal)
- a function may destroy the value of other registers e.g. $t0..t7$
- callers must assume value in e.g. $t0..t7$ changed by call (jal)
MIPS Register usage conventions - not covered in COMP1521

- floating point registers used to pass/return float/doubles
- similar conventions for saving floating point registers
- stack used to pass arguments after first 4
- stack used to pass arguments which do not fit in register
- stack used to return value which do not fit in register
- for example C argument or return value can be a struct, which is any number of bytes
some local (function) variables must be stored on stack

- e.g. variables such as arrays and structs

```c
int main(void) {
    int squares[10];
    int i = 0;
    while (i < 10) {
        squares[i] = i * i;
        i++;
    }
}
```

```assembly
main:
    sub $sp, $sp, 40
    li $t0, 0
    loop0:
    mul $t1, $t0, 4
    add $t2, $t1, $sp
    mul $t3, $t0, $t0
    sw $t3, ($t2)
    add $t0, $t0, 1
    b loop0
end0:
```
What is a Frame Pointer

- frame pointer $fp$ is a second register pointing to stack
- by convention set to point at start of stack frame
- provides a fixed point during function code execution
- useful for functions which grow stack (change $sp$) during execution
- makes it easier for debuggers to forensically analyze stack
- e.g. if you want to print stack backtrace after error
- frame pointer is optional (in COMP1521 and generally)
- often omitted when fast execution or small code a priority
Example of Growing Stack Breaking Function Return

```c
void f(int a) {
    int length;
    scanf("%d", &length);
    int array[length];
    // ... more code ...
}
```

```assembly
f:
    sub $sp, $sp, 4
    sw $ra, 0($sp)
    li $v0, 5
    syscall
    # allocate space for
    # array on stack
    mul $t0, $v0, 4
    sub $sp, $sp, $t0
    # ... more code ...
    # breaks because $sp
    # has changed
    lw $ra, 0($sp)
    add $sp, $sp, 4
    jr $ra
```
Example of Frame Pointer Use

```c
void f(int a) {
    int length;
    scanf("%d", &length);
    int array[length];
    // ... more code ...
}
```

```
f:
    sub  $sp, $sp, 8
    sw   $fp, 4($sp)
    sw   $ra, 0($sp)
    add  $fp, $sp, 8
    li   $v0, 5
    syscall
    mul  $t0, $v0, 4
    sub  $sp, $sp, $t0
    # ... more code ...
    lw   $ra, -4($fp)
    move $sp, $fp
    lw   $fp, 0($fp)
    jr   $ra
```