

# Data flow in C

- ❑ Functions accept **arguments** and produce **return values**.
- ❑ The **black** parts of the program show the actual and formal arguments of the fact function.
- ❑ The **purple** parts of the code deal with returning and using a result.

```
int main()
{
    ...
    t1 = fact(8);
    t2 = fact(3);
    t3 = t1 + t2;
    ...
}

int fact(int n)
{
    int i, f = 1;
    for (i = n; i > 1; i--)
        f = f * i;
    return f;
}
```

# Data flow in MIPS

---

- ❑ MIPS uses the following conventions for function arguments and results.
  - Up to four function arguments can be “passed” by placing them in argument registers **\$a0-\$a3** before calling the function with jal.
  - A function can “return” up to two values by placing them in registers **\$v0-\$v1**, before returning via jr.
  
- ❑ These conventions are not enforced by the hardware or assembler, but programmers agree to them so functions written by different people can interface with each other.
  
- ❑ Later we’ll talk about handling additional arguments or return values.

# Nested functions

- ❑ What happens when you call a function that then calls another function?
- ❑ Let's say A calls B, which calls C.
  - The arguments for the call to C would be placed in \$a0-\$a3, thus *overwriting* the original arguments for B.
  - Similarly, **jal C** overwrites the return address that was saved in \$ra by the earlier **jal B**.

```
A:    ...  
      # Put B's args in $a0-$a3  
      jal B      # $ra = A2  
A2:   ...
```

```
B:    ...  
      # Put C's args in $a0-$a3,  
      # erasing B's args!  
      jal C      # $ra = B2  
B2:   ...  
      jr $ra     # Where does  
                # this go???
```

```
C:    ...  
      jr $ra
```

# Spilling registers

---

- ❑ The CPU has a limited number of registers for use by all functions, and it's possible that several functions will need the same registers.
- ❑ We can keep important registers from being overwritten by a function call, by saving them before the function executes, and restoring them after the function completes.
- ❑ But there are two important questions.
  - Who is responsible for saving registers—the caller or the callee?
  - Where exactly are the register contents saved?



# Who saves the registers?

---

- ❑ **Who is responsible for saving important registers across function calls?**
  - The caller knows which registers are important to it and should be saved.
  - The callee knows exactly which registers it will use and potentially overwrite.
- ❑ **However, in the typical “black box” programming approach, the caller and callee do not know anything about each other’s implementation.**
  - Different functions may be written by different people or companies.
  - A function should be able to interface with any client, and different implementations of the same function should be substitutable.
- ❑ **So how can two functions cooperate and share registers when they don’t know anything about each other?**

# The caller could save the registers...

- ❑ One possibility is for the *caller* to save any important registers that it needs before making a function call, and to restore them after.
- ❑ But the caller does not know what registers are actually written by the function, so it may save more registers than necessary.
- ❑ In the example on the right, **frodo** wants to preserve **\$a0**, **\$a1**, **\$s0** and **\$s1** from **gollum**, but **gollum** may not even use those registers.

```
frodo: li    $a0, 3
        li    $a1, 1
        li    $s0, 4
        li    $s1, 1

        # Save registers
        # $a0, $a1, $s0, $s1

        jal   gollum

        # Restore registers
        # $a0, $a1, $s0, $s1

        add   $v0, $a0, $a1
        add   $v1, $s0, $s1
        jr    $ra
```

# ...or the callee could save the registers...

- ❑ Another possibility is if the *callee* saves and restores any registers it might overwrite.
- ❑ For instance, a **gollum** function that uses registers **\$a0**, **\$a2**, **\$s0** and **\$s2** could save the original values first, and restore them before returning.
- ❑ But the callee does not know what registers are important to the caller, so again it may save more registers than necessary.

```
gol l um:
```

```
# Save registers  
# $a0 $a2 $s0 $s2
```

```
li    $a0, 2  
li    $a2, 7  
li    $s0, 1  
li    $s2, 8
```

```
...
```

```
# Restore registers  
# $a0 $a2 $s0 $s2
```

```
jr    $ra
```

# ...or they could work together

- ❑ MIPS uses conventions again to split the register spilling chores.
- ❑ The *caller* is responsible for saving and restoring any of the following **caller-saved registers** that it cares about.

**\$t0-\$t9**

**\$a0-\$a3**

**\$v0-\$v1**

In other words, the callee may freely modify these registers, under the assumption that the caller already saved them if necessary.

- ❑ The *callee* is responsible for saving and restoring any of the following **callee-saved registers** that it uses. (Remember that \$ra is “used” by jal.)

**\$s0-\$s7**

**\$ra**

Thus the caller may assume these registers are not changed by the callee.

– **\$ra** is tricky; it is saved by a callee who is also a caller.

- ❑ Be especially careful when writing nested functions, which act as both a caller and a callee!

# Register spilling example

- This convention ensures that the caller and callee together save all of the important registers—frodo only needs to save registers \$a0 and \$a1, while gollum only has to save registers \$s0 and \$s2.

```
frodo:  li    $a0, 3
        li    $a1, 1
        li    $s0, 4
        li    $s1, 1

        # Save registers
        # $a0 and $a1

        jal   gollum

        # Restore registers
        # $a0 and $a1

        add   $v0, $a0, $a1
        add   $v1, $s0, $s1
        jr    $ra

gollum: # Save registers
        # $s0 and $s2

        li    $a0, 2
        li    $a2, 7
        li    $s0, 1
        li    $s2, 8
        ...

        # Restore registers
        # $s0 and $s2

        jr    $ra
```

# Where are the registers saved?

---

- ❑ **Now we know who is responsible for saving which registers, but we still need to discuss where those registers are saved.**
- ❑ **It would be nice if each function call had its own private memory area.**
  - **This would prevent other function calls from overwriting our saved registers—otherwise using memory is no better than using registers.**
  - **We could use this private memory for other purposes too, like storing local variables.**