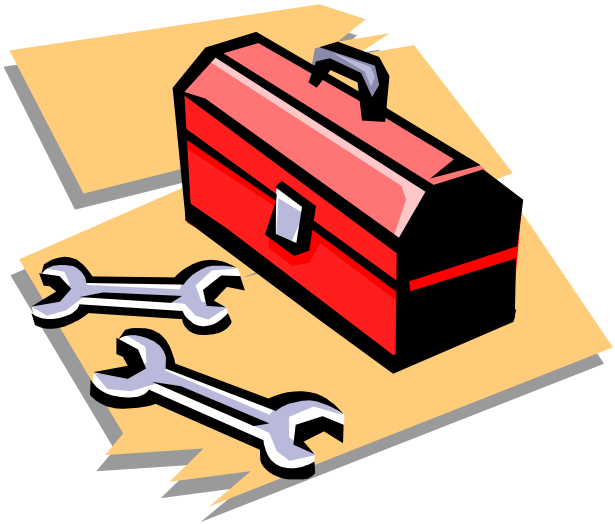
Functions in MIPS

- Function calls are relatively simple in a high-level language, but actually involve multiple steps and instructions at the assembly level.
  - The program's flow of control must be changed.
  - Arguments and returning values are passed back and forth.
  - Local variables can be allocated and destroyed.
- Today we'll see how these issues are handled in the MIPS architecture.
  - There are new instructions for calling functions.
  - Conventions are used for sharing registers between functions.
  - Functions can make good use of a stack in memory.



Control flow in C

- Invoking a function changes the control flow of a program twice.
  1. Calling the function
  2. Returning from the function
- In this example the main function calls `fact` twice, and fact returns twice—but to *different* locations in main.
- Each time fact is called, the CPU has to remember the appropriate return address.
- Notice that main itself is also a function! It is called by the operating system when you run the program.

```
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;
}
```

Control flow in MIPS

- MIPS uses the jump-and-link instruction jal to call functions.
  - The jal saves the return address (the address of the *next* instruction) in the dedicated register $ra, before jumping to the function.
  - jal is the only MIPS instruction that can access the value of the program counter, so it can store the return address PC+4 in $ra.

```
jal Fact
```

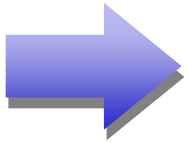
- To transfer control back to the caller, the function just has to jump to the address that was stored in $ra.

```
jr $ra
```

- The code on the next page shows the jal and jr instructions that are necessary for our factorial example.

Control flow in the example

```
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;
}
```



```
main:
        ...
        jal fact
L1:     ...
        jal fact
L2:     ...
        ...
        jr  $ra

fact:
        ...
        ...
        ...
        ...
        ...
        jr  $ra
```

Data flow in C

- Functions accept arguments and produce return values.
- The blue 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 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.

Data flow in the example: fact

- The fact function has only one argument and returns just one value.
- The blue assembly code shows the function using its argument, which should have been placed in $a0 by the caller.
- The purple instructions show fact putting a return value in $v0 before giving control back to the caller.
- Register $t0 represents local variable f, and register $t1 represents local variable i.

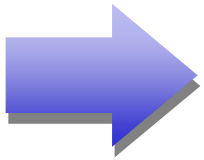
```
int fact(int n)
{
  int i, f = 1;
  for (i = n; i > 1; i--)
    f = f * i;
  return f;
}
```

```
fact:
    li    $t0, 1          # f = 1
    move  $t1, $a0        # i = n
loop:
    ble   $t1, 1, ret     # i > 1
    mul   $t0, $t0, $t1   # f = f x i
    sub   $t1, $t1, 1     # i--
    j     loop
ret:
    move  $v0, $t0        # return f
    jr    $ra
```

Data flow in the example: main

- The blue MIPS code shows main passing the actual parameters 8 and 3, by placing them in register $a0 before the jal instructions.
- The purple lines show how the function result in register $v0 can then be accessed by the caller—here for storage into $t1 and $t2.

```
int main()
{
   ...
   t1 = fact(8);



   t2 = fact(3);


   t3 = t1 + t2;
   ...
}
```

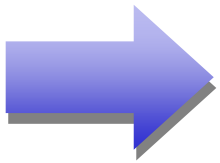


```
main:
      ...
      li    $a0, 8
      jal   fact
      move  $t1, $v0

      li    $a0, 3
      jal   fact
      move  $t2, $v0

      add   $t3, $t1, $t2
      ...
      jr    $ra
```

A note about optimization

- We could actually save a couple of instructions in this code.
  - Instead of moving the result $t0 into $v0 at the end of the function, we could just use $v0 throughout the function.
  - Similarly, we could use register $a0 without first copying it into $t1.
- We'll use the unoptimized version to illustrate some other points.

```
fact:
     li    $t0, 1
     move  $t1, $a0
loop:
     ble   $t1, 1, ret
     mul   $t0, $t0, $t1
     sub   $t1, $t1, 1
     j     loop
ret:
     move  $v0, $t0
     jr    $ra
```



```
fact:
     li    $v0, 1

loop:
     ble   $a0, 1, ret
     mul   $v0, $v0, $a0
     sub   $a0, $a0, 1
     j     loop

ret:
     jr    $ra
```

A note about types

- Assembly language is untyped—there is no distinction between integers, characters, pointers or other kinds of values.
- It is up to *you* to typecheck your programs. In particular, make sure your function arguments and return values are used consistently.
- For example, what happens if somebody passes the *address* of an integer (instead of the integer itself) to the fact function?

```
fact:
      li   $t0, 1
      move $t1, $a0
loop:
      ble  $t1, 1, ret
      mul  $t0, $t0, $t1
      sub  $t1, $t1, 1
      j    loop
ret:
      move $v0, $t0
      jr   $ra
```

The big problem so far

- There is a big problem here!
  - The main code uses $t1 to store the result of fact(8).
  - But $t1 is also used within the fact function!
- The subsequent call to fact(3) will overwrite the value of fact(8) that was stored in $t1.

```
main:  li    $a0, 8
       jal   fact
       move  $t1, $v0
       li    $a0, 3
       jal   fact
       move  $t2, $v0
       add   $t3, $t1, $t2
       jr    $ra

fact:  li    $t0, 1
       move  $t1, $a0
loop:  ble   $t1, 1, ret
       mul   $t0, $t0, $t1
       sub   $t1, $t1, 1
       j     loop
ret:   move  $v0, $t0
       jr    $ra
```

Nested functions

- A similar situation 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.

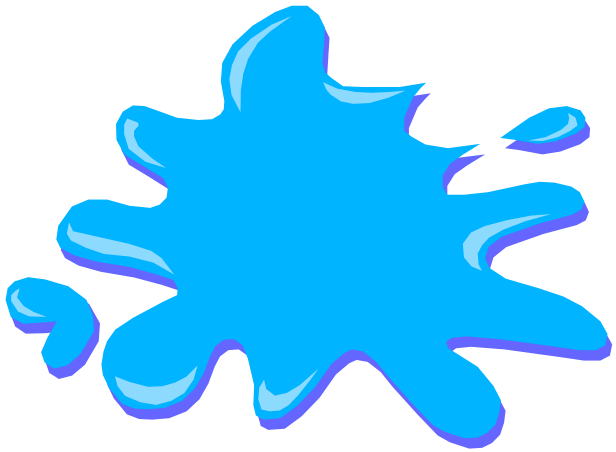
```
A:   ...
     # Put B's args in $a0-$a3
     jal  B       # $ra = A2
A2:  ...
```

```
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.

```
gollum:
        # 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.
- 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
```

How to fix factorial

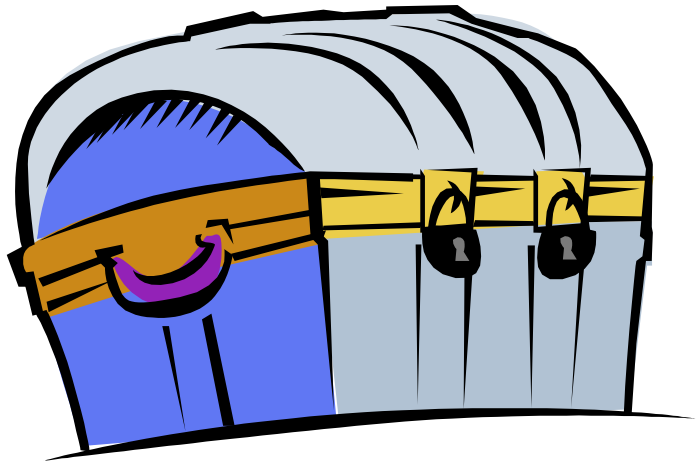
- In the factorial example, main (the caller) should save two registers.
  - $t1 must be saved before the second call to fact.
  - $ra will be implicitly overwritten by the jal instructions.
- But fact (the callee) does not need to save anything. It only writes to registers $t0, $t1 and $v0, which should have been saved by the caller.

```
main:
        #--Save $ra--
        li      $a0, 8
        jal     fact
        move    $t1, $v0
        #--Save $t1--
        li      $a0, 3
        jal     fact
        move    $t2, $v0
        #--Restore $t1--
        add     $t3, $t1, $t2
        #--Restore $ra--
        jr      $ra
```

```
fact:
        li      $t0, 1
        move    $t1, $a0
loop:
        ble     $t1, 1, ret
        mul     $t0, $t0, $t1
        sub     $t1, $t1, 1
        j       loop
ret:
        move    $v0, $t0
        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.

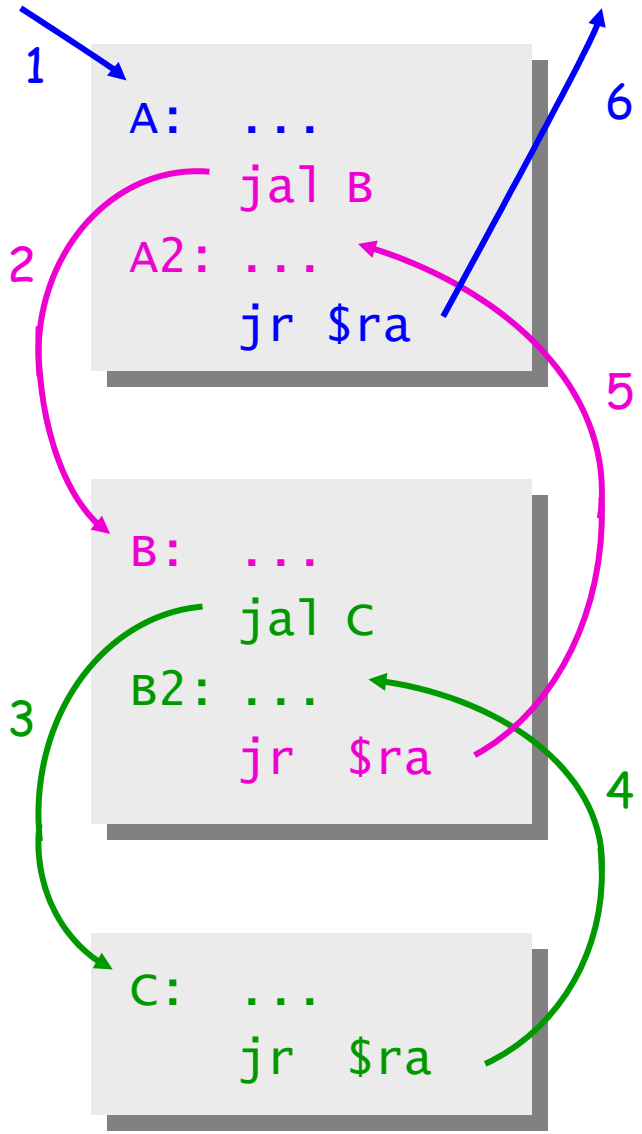


Function calls and stacks

- 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.

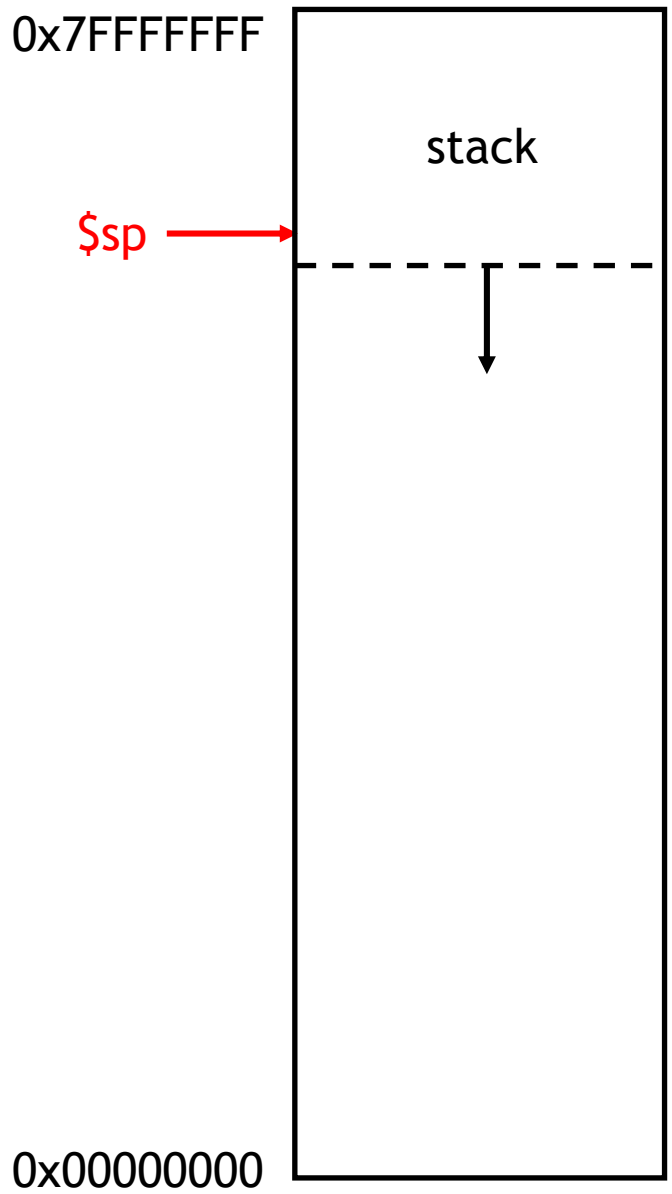


Stacks and function calls

- It's natural to use a stack for function call storage. A block of stack space, called a stack frame, can be allocated for each function call.
  - When a function is called, it creates a new frame onto the stack, which will be used for local storage.
  - Before the function returns, it must pop its stack frame, to restore the stack to its original state.
- The stack frame can be used for several purposes.
  - Caller- and callee-save registers can be put in the stack.
  - The stack frame can also hold local variables, or extra arguments and return values.

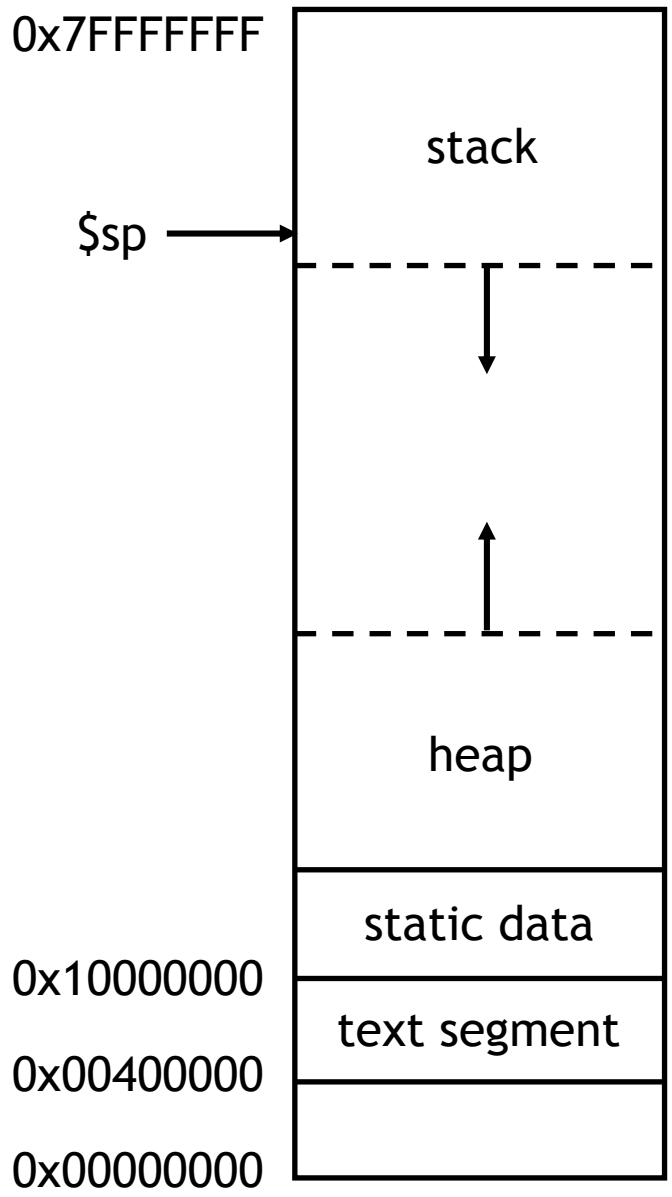
The MIPS stack

- In MIPS machines, part of main memory is reserved for a stack.
  - The stack grows downward in terms of memory addresses.
  - The address of the top element of the stack is stored in yet another dedicated register, $sp (stack pointer).
- MIPS does not provide "push" and "pop" instructions. Instead, they must be done explicitly by the programmer.



MIPS memory usage

- What goes into the rest of MIPS memory?
- A heap stores dynamically allocated data.
  - It grows upwards, toward the stack.
  - This lets the stack and heap each grow as large as necessary.
- Static data holds mostly global variables.
- The text segment contains your program code and serves as the instruction memory.
- You can see each of these areas in the main window when you run SPIM.



Pushing elements

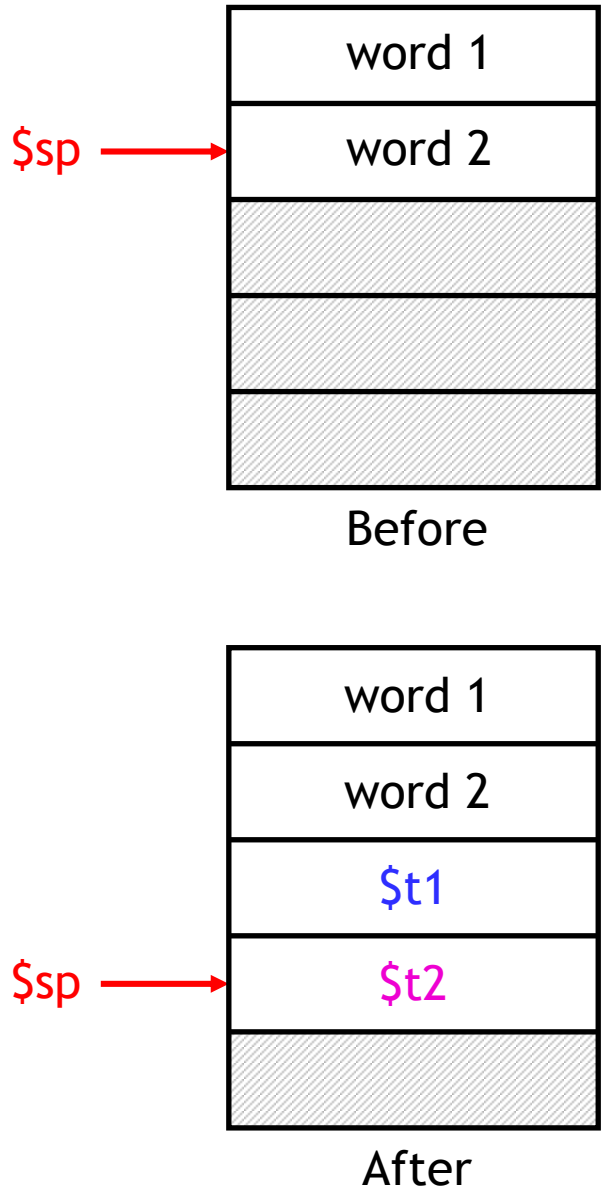
- To push elements onto the stack:
  - Move the stack pointer $sp down to make room for the new data.
  - Store the elements into the stack.
- For example, to push registers $t1 and $t2 onto the stack:

```
sub $sp, $sp, 8
sw  $t1, 4($sp)
sw  $t2, 0($sp)
```

- An equivalent sequence is:

```
sw  $t1, -4($sp)
sw  $t2, -8($sp)
sub $sp, $sp, 8
```

- Before and after diagrams of the stack are shown on the right.



Accessing and popping elements

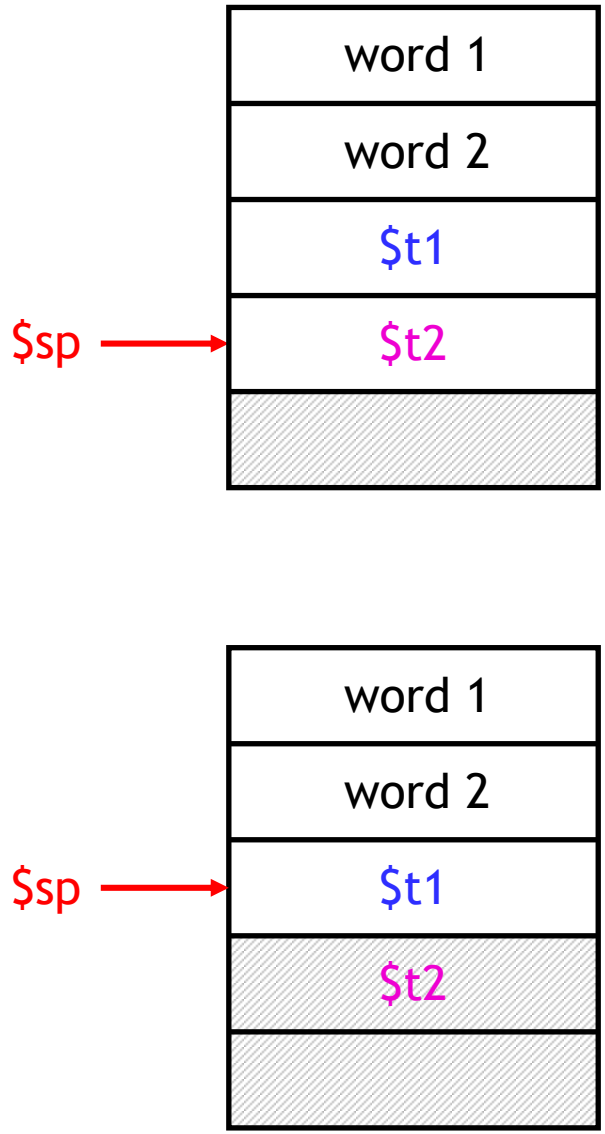
- You can access any element in the stack (not just the top one) if you know where it is relative to $sp.
- For example, to retrieve the value of $t1:

```
lw   $s0, 4($sp)
```

- You can pop, or "erase," elements simply by adjusting the stack pointer upwards.
- To pop the value of $t2, yielding the stack shown at the bottom:

```
addi $sp, $sp, 4
```

- Note that the popped data is still present in memory, but data past the stack pointer is not valid.



The example one last time

- The main code needs two words of stack space—$t1 is stored at 0($sp), and $ra is stored at 4($sp).
- It's easiest to adjust $sp once at the beginning and once at the end.

```
main:
        sub   $sp, $sp, 8    # Allocate two words on stack
        sw    $ra, 4($sp)    # Save $ra because of jal

        li    $a0, 8
        jal   fact
        move  $t1, $v0

        sw    $t1, 0($sp)    # Save $t1 for later use

        li    $a0, 3
        jal   fact
        move  $t2, $v0

        lw    $t1, 0($sp)    # Restore $t1

        add   $t3, $t1, $t2

        lw    $ra, 4($sp)    # Restore $ra
        addi  $sp, $sp, 8    # Deallocate stack frame

        jr    $ra
```

Summary

- Today we focused on implementing function calls in MIPS.
  - We call functions using jal, passing arguments in registers $a0-$a3.
  - Functions place results in $v0-$v1 and return using jr $ra.
- Managing resources is an important part of function calls.
  - To keep important data from being overwritten, registers are saved according to conventions for caller-save and callee-save registers.
  - Each function call uses stack memory for saving registers, storing local variables and passing extra arguments and return values.
- MIPS programmers must follow many conventions. Nothing prevents a rogue program from overwriting registers or stack memory used by some other function.
- Next time we'll look at more example programs, some of which even involve recursion!