MIPS and SPIM tutorial

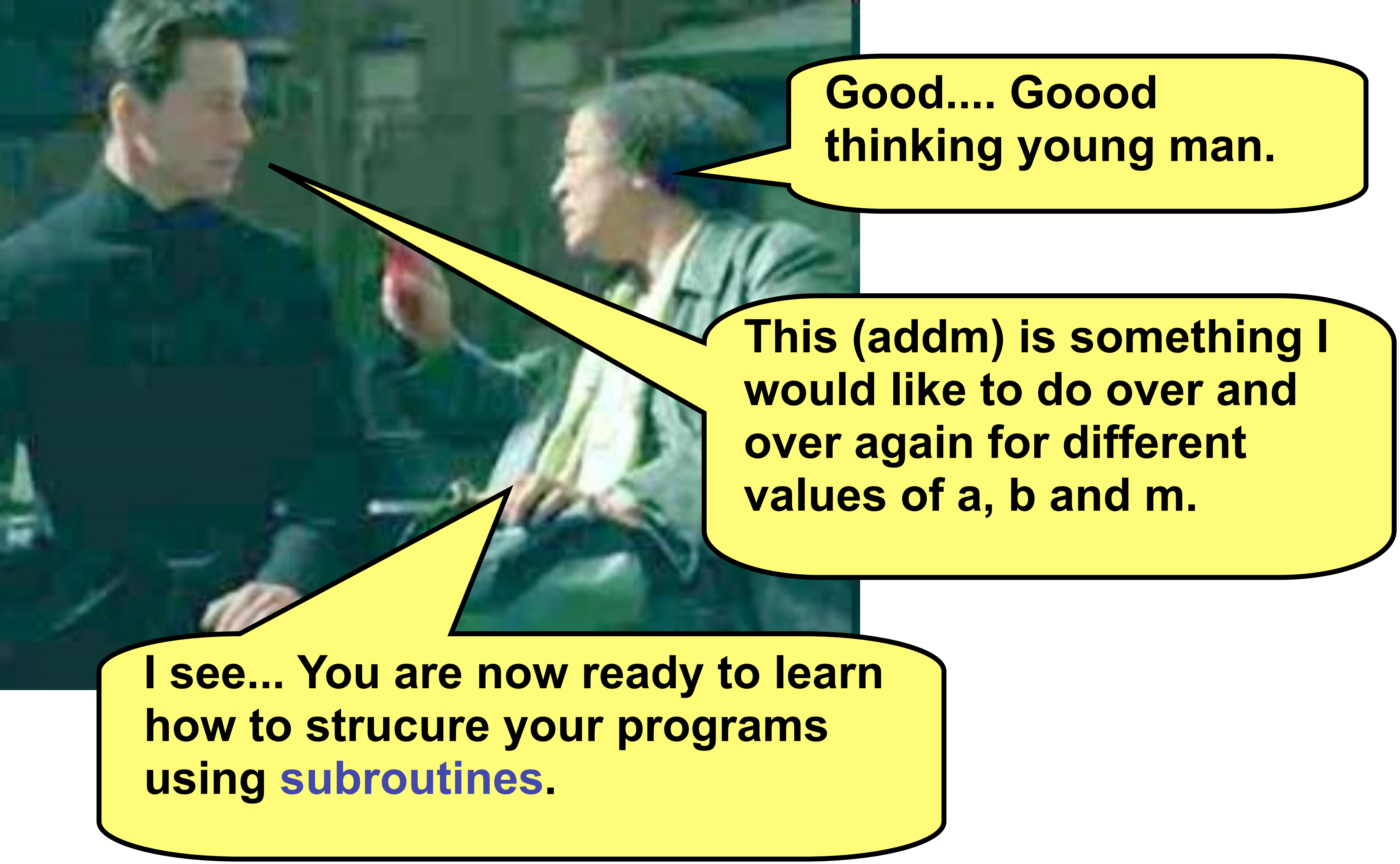
Part Three: Subroutines, jal, jr and the stack (push & pop)

November 2009

karl.marklund@it.uu.se

Get ready for part three of your MIPS assembly programming training.


```
# Calculate the sum a + b  and store the result to memory location m.

addm     m, a, b                  # But no such such instruction exist
```




```
# Calculate the sum a + b  and store the result to memory location m.

addm     m, a, b                    # But no such such instruction exist
```

Start by describing your subroutine:

```
##############################################################################
#
#  DESCRIPTION: Calculate the sum S = A + B an
#               to memory location M.
#
#        INPUT: $a0 - M (addresss)
#               $a1 - A (integer)
#               $a2 - B (integer)
#
#       OUTPUT: $v0 - S (integer)
#
# SIDE EFFECTS: The the sum S is written to the memory location M.
#
##############################################################################
```

Input is allways in $a-registers

Output is allways in a $v0-register

Any side effects must be stated.

Introduce *abstractions*

Don’t use register names in your comments.

```
###############################################################################
#
#  DESCRIPTION: Calculate the sum S = A + B and store the result
#               to memory location M.
#
#        INPUT: $a0 - M (addresss)
#               $a1 - A (integer)
#               $a2 - B (integer)
#
#       OUTPUT: $v0 - S (integer)
#
# SIDE EFFECTS: The the sum S is written to the memory location M.
#
###############################################################################
```

Refer to the previously introduced abstractions in your comments.

Any *side effects* must be stated.

```
##############################################################################
#
#  DESCRIPTION: Calculate the sum S = A + B and store the result
#               to memory location M.
#
#        INPUT: $a0 - M (addresss)
#               $a1 - A (integer)
#               $a2 - B (integer)
#
#       OUTPUT: $v0 - S (integer)
#
# SIDE EFFECTS: The the sum S is written to the      y locat
#
##############################################################
addm:
        add     $v0, $a1, $a2                   # S = A + B
        sw      $v0, 0($a0)                     # Store S to memory at address M

        jr      $ra                             # Return to caller
```

Now you can write the code to implement the desired functionality.

Refer to the previously introduced abstractions in your comments.

Give the subroutine a name – a label

Allways end a subroutine with jr $ra (jump register $ra)

```
##############################################################################
#
#  DESCRIPTION: Calculate the sum S = A + B and store the result
#               to memory location M.
#
#        INPUT: $a0 - M (addresss)
#               $a1 - A (integer)
#               $a2 - B (integer)
#
#       OUTPUT: $v0 - S (integer)
#
# SIDE EFFECTS: The the sum S is written to the memory location M.
#
##############################################################################
addm:
        add     $v0, $a1, $a2           # S = A + B
        sw      $v0, 0($a0)             # Store S to memory at address M

        jr      $ra                     # Return to caller
```

Writing *good comments* and *good looking code* does matter!

Use *tabs* to align your code.

Align your comments.

```
        .data

X:      .word 0x11111111
Y:      .word 0x22222222

        .text
                .globl main

main:   la      $a0, X              # Address of X
        li      $a1, 127            # a = 127
        li      $a2, 0xa            # b = 0xa = 10 (decimal)

        jal     addm                # store s =  a + b in memory at
                                    # address X

        la      $a0, Y              # Address of Y
        add     $a1, $v0,$zero      # a = s
        li      $a2, -3             # b = -3
        jal     addm                # store a + b in memory at
                                    # address Y

        nop

        jr      $ra
```

```
        .data

X:      .word 0x11111111
Y:      .word 0x22222222

        .text
                .globl main

main:   la      $a0, X           # Address of X
        li      $a1, 127         # a = 127
        li      $a2, 0xa         # b = 0xa = 10 (decimal)

                addm             # st                  memory at
                                 #

                                   Ad
                                   a =
                                   b = -3
                                   store a + b in memory at
                                 # address Y

        nop

        jr      $ra
```

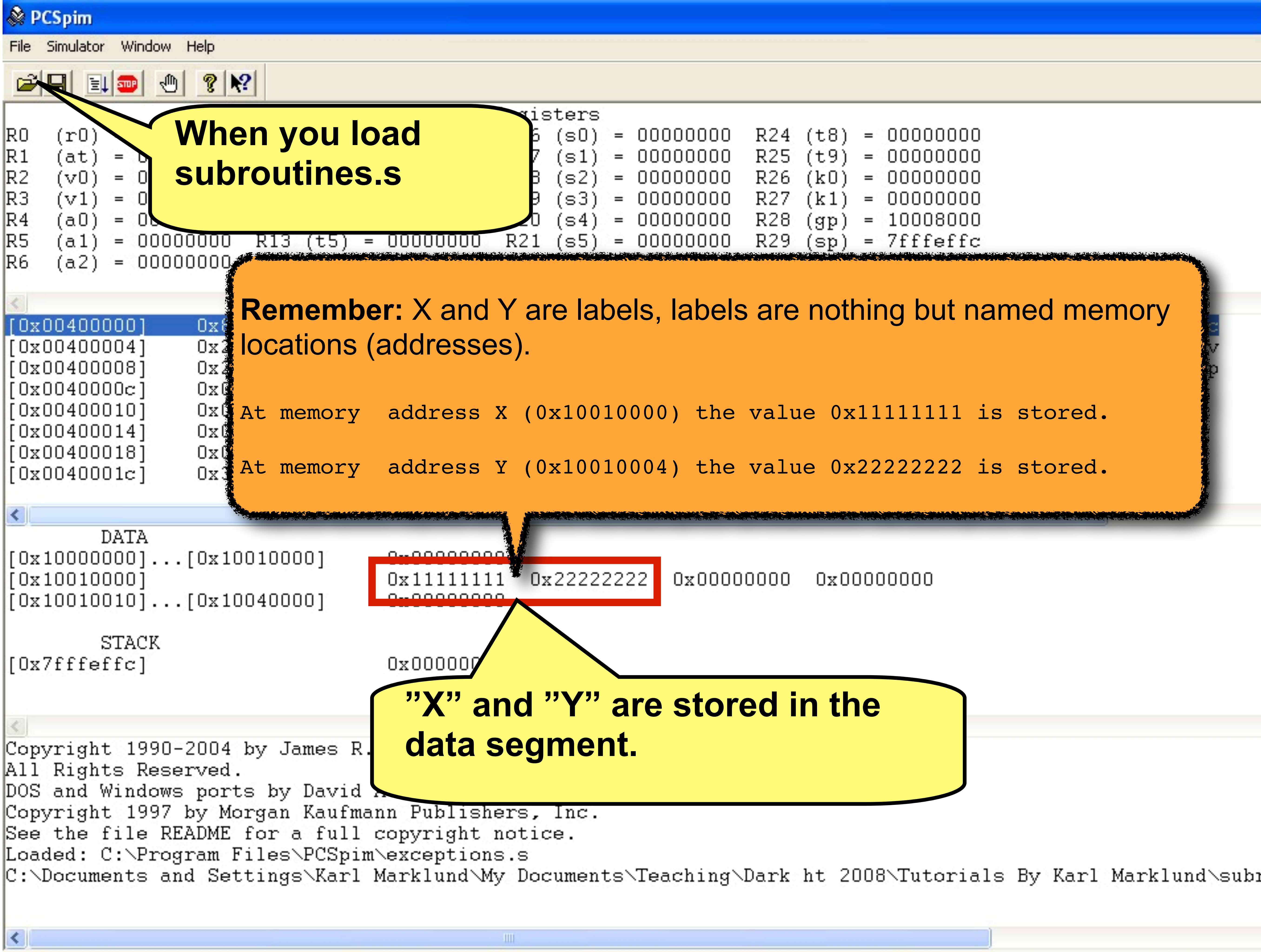
To test the subroutine we store these values in the data segment

Introduce and reuse abstractions in your comments.

Can use decimal or hexadecimal notation.

Load Immediate (li)

Set the content of register $a2 to $10_{10} = a_{16} = 0xa$ using a immediate constant.



When you load subroutines.s

Remember: X and Y are labels, labels are nothing but named memory locations (addresses).

`At memory address X (0x10010000) the value 0x11111111 is stored.`

`At memory address Y (0x10010004) the value 0x22222222 is stored.`

”X” and ”Y” are stored in the data segment.

```
        .data

X:      .word 0x11111111
Y:      .word 0x22222222

        .text
        .globl main

main:   la      $a0, X          # Address of X
        li      $a1, 127        # a = 127
        li      $a2, 0xa        # b = 0xa = 10 (decimal)

        jal     addm            # store s =  a + b in memory at address X

$ra →   la      $a0, Y          # Address of Y
        add     $a1, $v0,
        li      $a2, -3
                addm            emor   ddress Y
```

Prepare the call to the subroutine by providing input in the $a-registers.

Jump and Link to the subroutine.

Jump and Link stores the address to the instruction following the jal in $ra.

The execution will now continue in the subroutine...

```
##############################################################################
#
#  DESCRIPTION: Calculate the sum S = A + B and store the result
#               to memory location M.
#
#       INPUT: $a0 - M (addresss)
#               $a1 - A (integer)
#               $a2 - B (integer)
#
#      OUTPUT: $v0 - S (integer)
#
# SIDE EFFECTS: The the sum S is written to the memory location M.
#
###############################################################################
addm:
PC →     add      $v0, $a1, $a2                # S = A + B
         sw       $v0, 0($a0)                  # Store S to memory at address M

         jr       $ra                          # Return to caller
```

```
##############################################################################
#
#  DESCRIPTION: Calculate the sum S = A + B and store the result
#               to memory location M.
#
#        INPUT: $a0 - M (addresss)
#               $a1 - A (integer)
#               $a2 - B (integer)
#
#       OUTPUT: $v0 - S (integer)
#
# SIDE EFFECTS: The the sum S is written to the memory location M.
#
###############################################################################
addm:
        add     $v0, $a1, $a2                 # S = A + B
PC →    sw      $v0, 0($a0)                   # Store S to memory at address M

        jr      $ra                           # Return to caller
```

```
################################################################################
#
#  DESCRIPTION: Calculate the sum S = A + B and store the result
#               to memory location M.
#
#        INPUT: $a0 - M (addresss)
#               $a1 - A (integer)
#               $a2 - B (integer)
#
#       OUTPUT: $v0 - S (integer)
#
# SIDE EFFECTS: The the sum S is written to the memory location M.
#
################################################################################
addm:
        add      $v0, $a1, $a2                   # S = A + B
        sw       $v0, 0($a0)                     # Store S to memory at address M

PC →    jr       $ra                             # Return to caller
```

We can return to the caller using the stored return address in $ra

The sum S is *returned* in the output register $v0.

```
        .data

X:      .word 0x11111111
Y:      .word 0x22222222

        .text
        .globl main

main:   la      $a0, X              # Address of X
        li      $a1, 127            # a = 127
        li      $a2, 0xa            # b = 0xa = 10 (decimal)

        jal     addm                # store s =  a + b in memory at address X

$ra →   la      $a0, Y              # Address of Y
        add     $a1, $v0,$zero      # a = S
        li      $a2, -3             # b = -3
        jal     addm                # store a + b in memory at address Y

        nop

        jr      $ra
```

```
        .data

X:      .word 0x11111111
Y:      .word 0x22222222

        .text
        .globl main

main:   la      $a0, X              # Address of X
        li      $a1, 127            # a = 127
        li      $a2, 0xa            # b = 0xa = 10 (decimal)

        jal     addm                # store s =  a + b in memory at address X

PC →    la      $a0, Y              # Address of Y
        add     $a1, $v0,$zero      # a = S
        li      $a2, -3             # b = -3
        jal     addm                # store a + b in memory at address Y

        nop

        jr      $ra
```

```
        .data

X:      .word 0x11111111
Y:      .word 0x22222222

        .text
        .globl main

main:   la      $a0, X              # Address of X
        li      $a1, 127            # a = 127
        li      $a2, 0xa            # b = 0xa = 10 (de

        jal     addm                # store s =  a + b in

        la      $a0, Y              # Address of Y
        add     $a1, $v0,$zero      # a = S
        li      $a2, -3             # b = -3

        jal     addm                # store a + b in memory at address Y

$ra →   nop

        jr      $ra
```

Prepare a 2nd call to the subroutine by providing input in the $a-registers.

Jump and Link stores the address to the instruction following the jal in $ra.

```
###############################################################################
#
#  DESCRIPTION: Calculate the sum S = A + B and store the result
#               to memory location M.
#
#        INPUT: $a0 - M (addresss)
#               $a1 - A (integer)
#               $a2 - B (integer)
#
#       OUTPUT: $v0 - S (integer)
#
# SIDE EFFECTS: The the sum S is written to the memory location M.
#
################################################################################
addm:
PC →    add      $v0, $a1, $a2                 # S = A + B
        sw       $v0, 0($a0)                   # Store S to memory at address M

        jr       $ra                           # Return to caller
```

```
##############################################################################
#
#  DESCRIPTION: Calculate the sum S = A + B and store the result
#               to memory location M.
#
#        INPUT: $a0 - M (addresss)
#               $a1 - A (integer)
#               $a2 - B (integer)
#
#       OUTPUT: $v0 - S (integer)
#
# SIDE EFFECTS: The the sum S is written to the memory location M.
#
###############################################################################
addm:
         add       $v0, $a1, $a2                   # S = A + B
 PC →    sw        $v0, 0($a0)                     # Store S to memory at address M

         jr        $ra                             # Return to caller
```

```
##############################################################################
#
#  DESCRIPTION: Calculate the sum S = A + B and store the result
#               to memory location M.
#
#        INPUT: $a0 - M (addresss)
#               $a1 - A (integer)
#               $a2 - B (integer)
#
#       OUTPUT: $v0 - S (integer)
#
# SIDE EFFECTS: The the sum S is written to the memory location M.
#
###############################################################################
addm:
         add      $v0, $a1, $a2                 # S = A + B
         sw       $v0, 0($a0)                   # Store S to memory at address M

 PC →    jr       $ra                           # Return to caller
```

```
        .data

X:      .word 0x11111111
Y:      .word 0x22222222

        .text
        .globl main

main:   la      $a0, X              # Address of X
        li      $a1, 127            # a = 127
        li      $a2, 0xa            # b = 0xa = 10 (decimal)

        jal     addm                # store s =  a + b in memory at address X

        la      $a0, Y              # Address of Y
        add     $a1, $v0,$zero      # a = s
        li      $a2, -3             # b = -3
        jal     addm                # store a + b in memory at address Y

$ra →   nop

        jr      $ra
```

No Operation – nop

An instruction that does nothing.

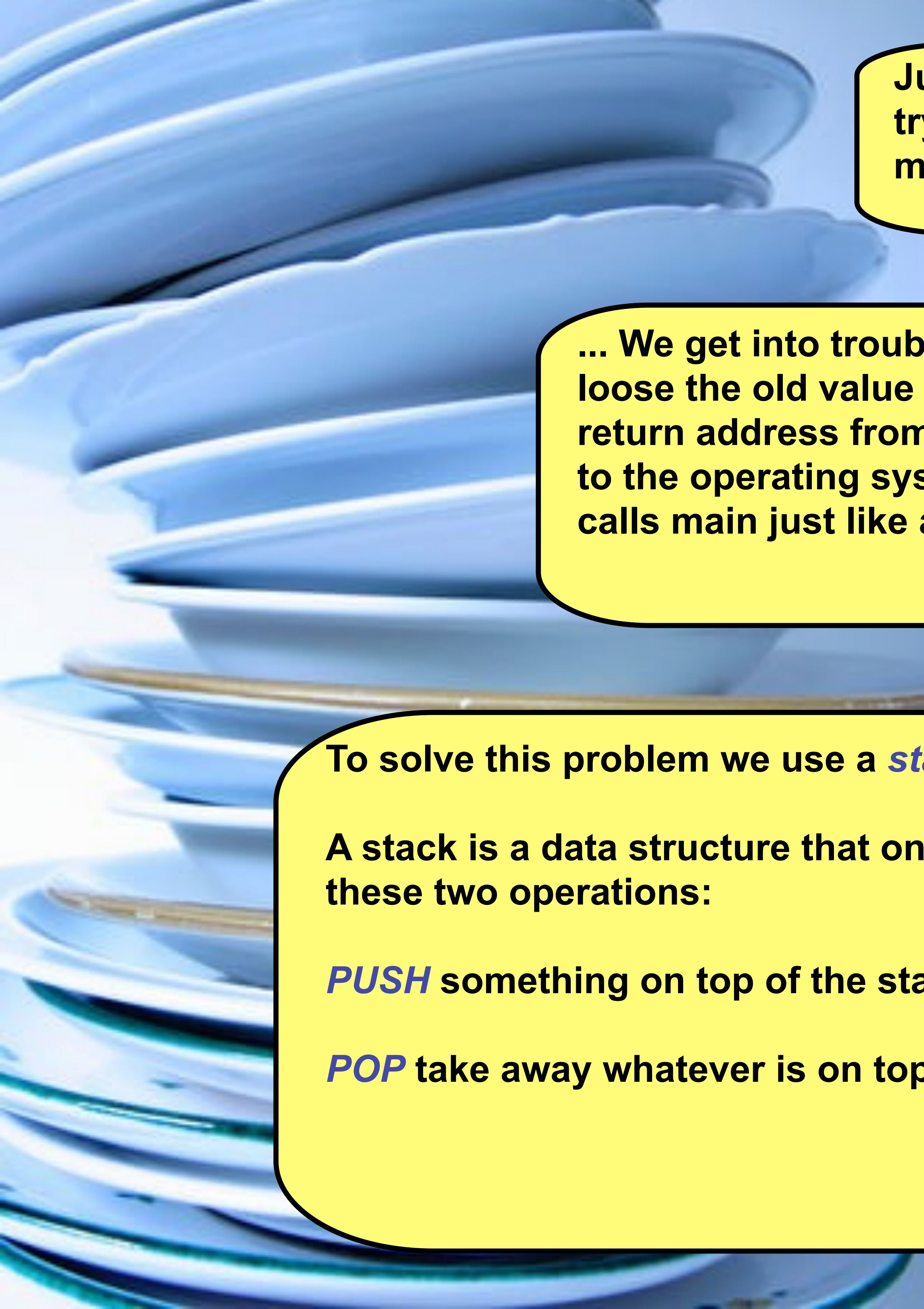
```
        .data

X:      .word 0x11111111
Y:      .word 0x22222222

        .text
        .globl main

main:   la      $a0, X              # Address of X
        li      $a1, 127            # a = 127
        li      $a2, 0xa            # b = 0xa = 10 (decimal)

        jal     addm                # store s =  a + b in memory at address X

        la      $a0, Y              # Address of Y
        add     $a1, $v0,$zero      # a = s
        li      $a2, -3             # b = -3
        jal     addm                # store a + b in memory at address Y

$PC →   nop

        jr      $ra
```

```
        .data

X:      .word 0x11111111
Y:      .word 0x22222222

        .text
        .globl main

main:   la      $a0, X            # Address of X
        li      $a1, 127          # a = 127
        li      $a2, 0xa          # b = 0xa = 10 (decimal)

        jal     addm              # store s =  a + b in memory at address X

        la      $a0, Y            # Address of Y
        add     $a1, $v0,$zero    # a = s
        li      $a2, -3           # b = -3
        jal     addm              # store a + b in memory at address Y

        nop

$PC →   jr      $ra
```

```
        .data

X:      .word 0x11111111
Y:      .word 0x22222222

        .text
        .globl main

main:   la      $a0, X              # Address of X
        li      $a1, 127            # a = 127
        li      $a2, 0xa            # b = 0xa = 10 (decimal)

        jal     addm                # store s =  a + b in memory at address X

        la      $a0, Y              # Address of Y
        add     $a1, $v0,$zero      # a = s
        li      $a2, -3             # b = -3
        jal     addm                # store a + b in memory at address Y

$PC →   nop

        jr      $ra
```

```
        .data

X:      .word 0x11111111
Y:      .word 0x22222222

        .text
        .globl main

main:   la      $a0, X              # Address of X
        li      $a1, 127            # a = 127
        li      $a2, 0xa            # b = 0xa = 10 (decimal)

        jal     addm                # store s =  a + b in memory at address X

        la      $a0, Y              # Address of Y
        add     $a1, $v0,$zero      # a = s
        li      $a2, -3             # b = -3
        jal     addm                # store a + b in memory at address Y

$ra →   nop

$PC →   jr      $ra
```

We are stuck in an infinite loop since $ra points here.



Just like it’s a bad idea to try to pick a plate in the middle of the pile...

... We get into trouble when we loose the old value in $ra, the return address from main back to the operating system (who calls main just like a subroutine).

To solve this problem we use a *stack.*

A stack is a data structure that only allows these two operations:

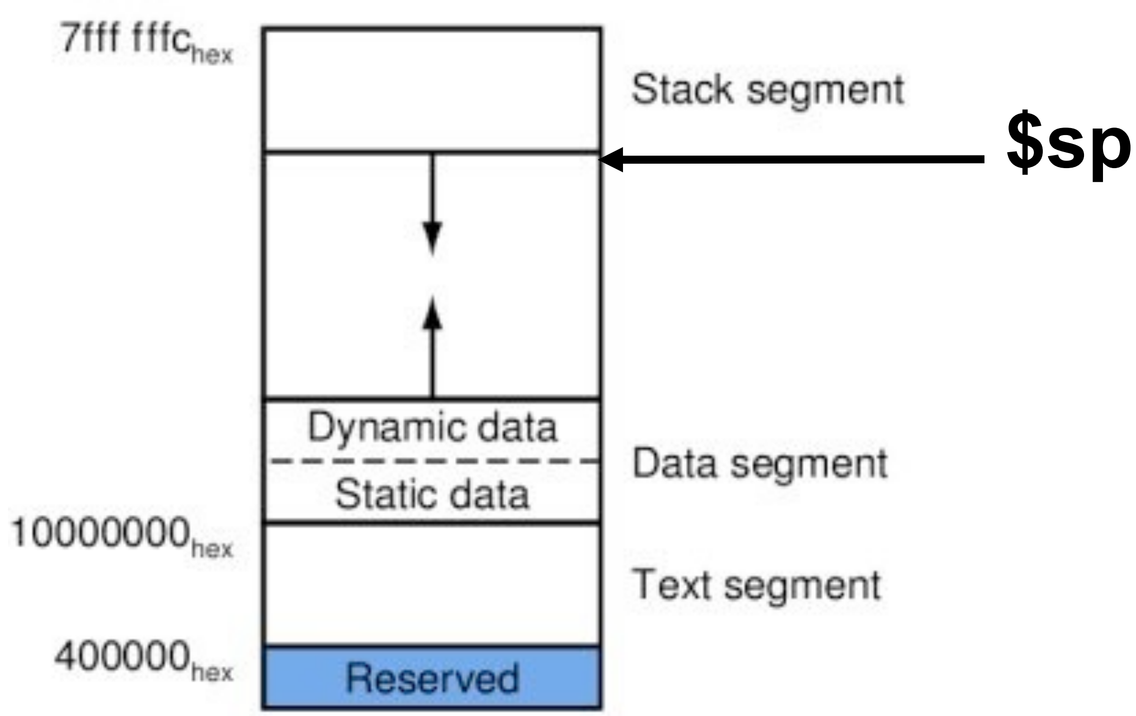
PUSH something on top of the stack.

POP take away whatever is on top of the stack.

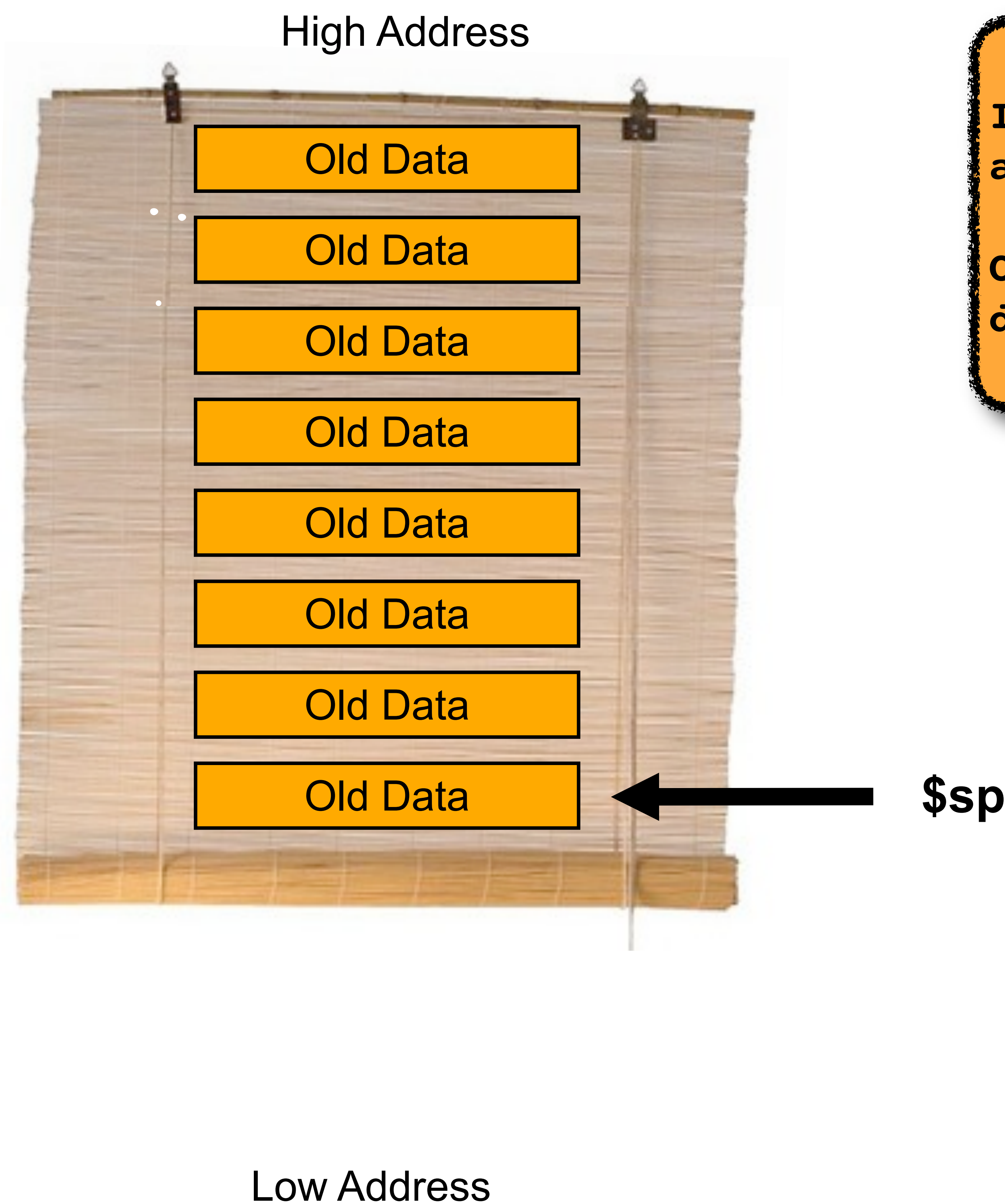
In MIPS the special *$sp* register is used to define the top of the stack.

The stacks grows downward from high addresses to low addresses:

This is to not interfere with the data segment.



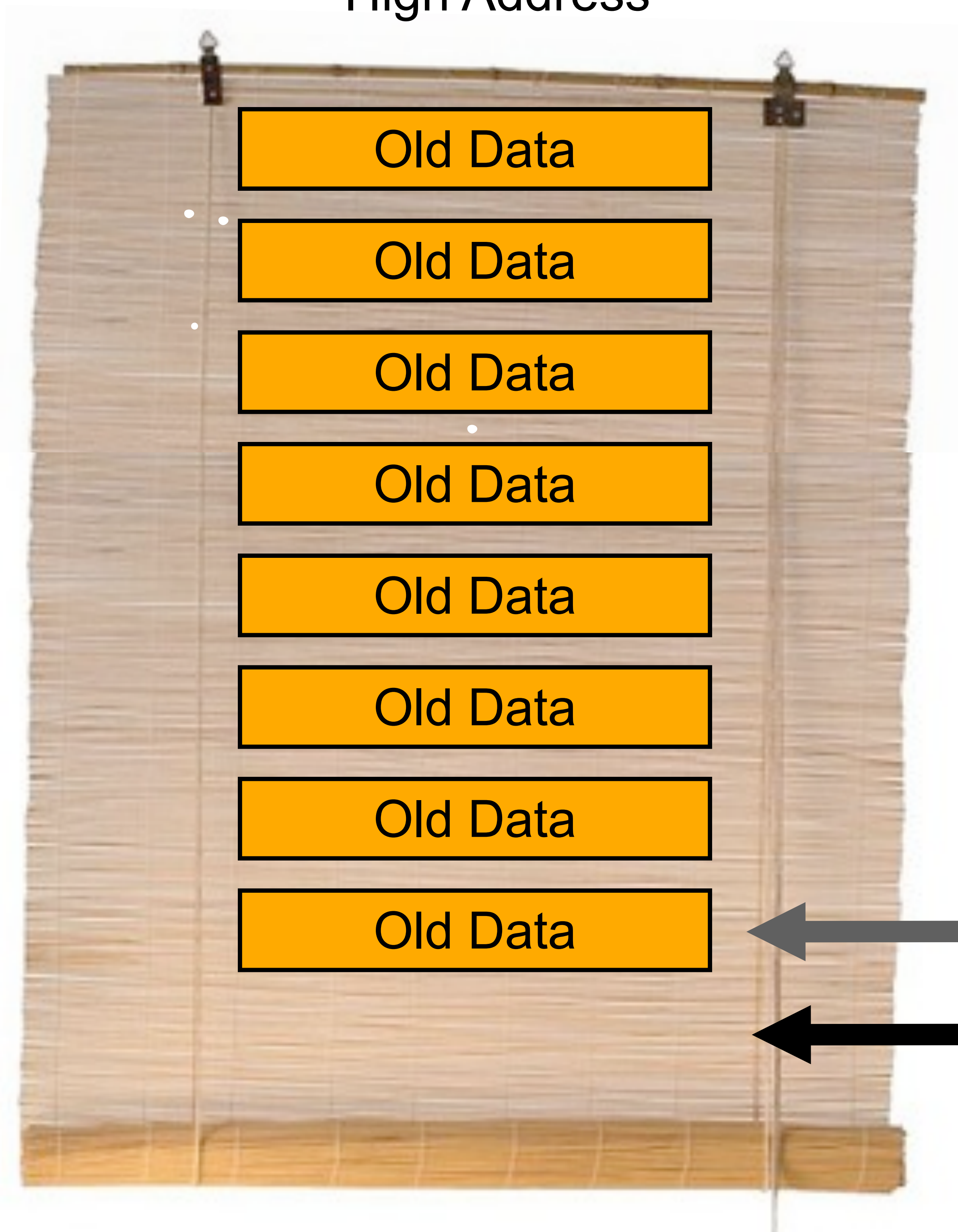

```
# PUSH register $t0

addi    $sp, $sp, -4
sw      $t0, 0($sp)


# POP value on top of stack to $t0

lw      $t0, 0($sp)
addi    $sp, $sp, 4
```



In MIPS, the stack is like a pull down curtain.

On the curtain we write data words (4 bytes).

High Address



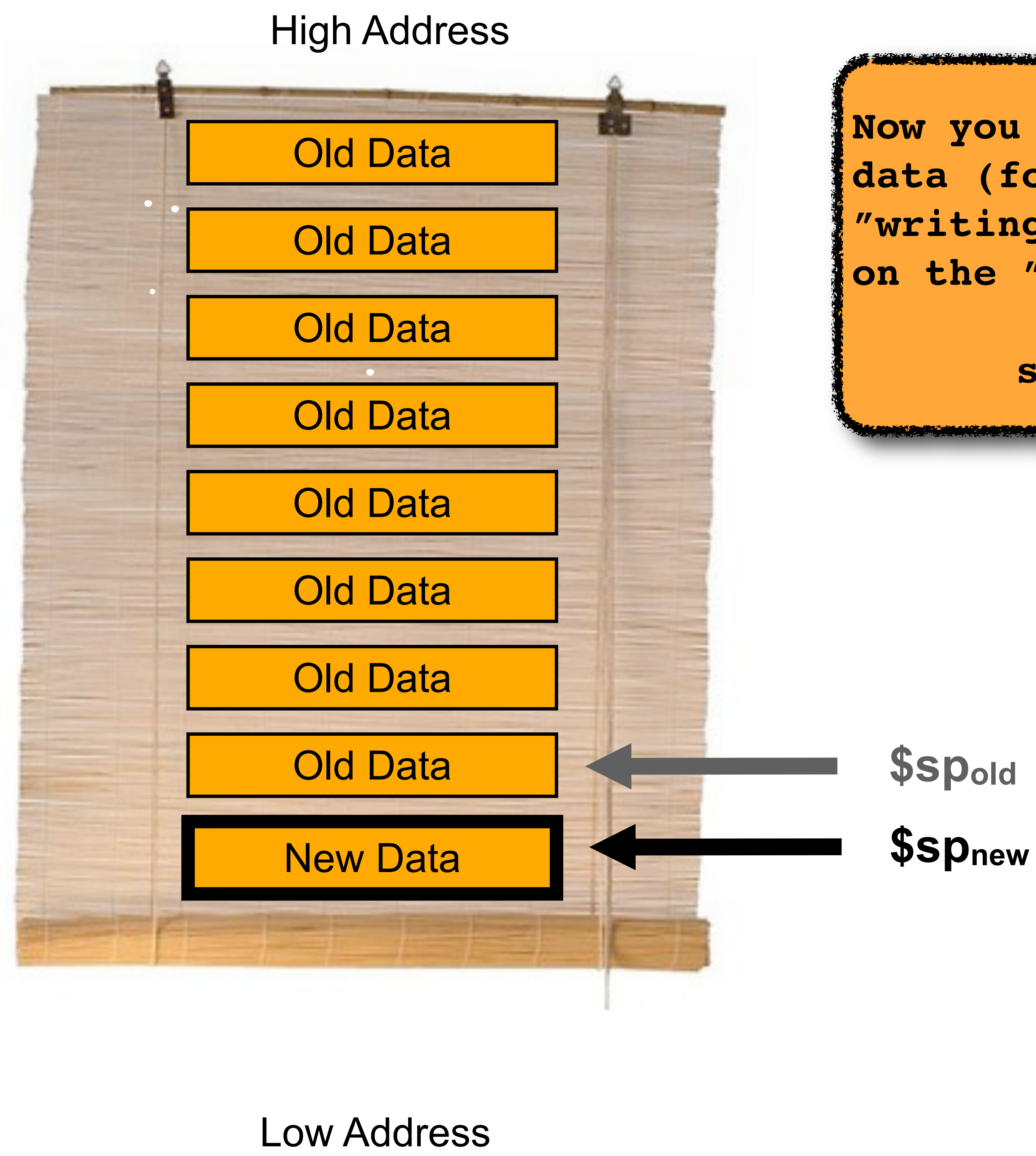
Low Address

To PUSH new data on the stack, pull the "curtain" down to make space for the new data.

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

$\$sp_{old}$

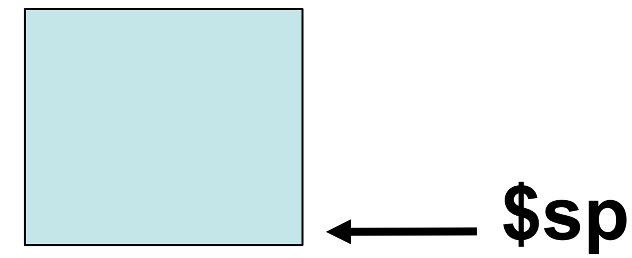
$\$sp_{new}$



Now you can store the new data (for example $t0) by "writing" to the new space on the "curtain"

```
sw $t0, 0($sp)
```

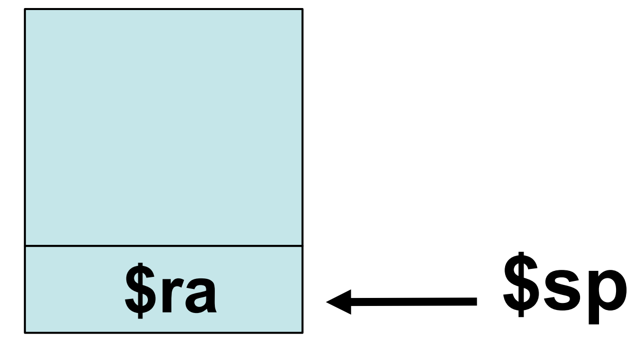
In the beginning of main we store the return address (back to the caller of main) on the stack (push).



Before the PUSH

```
        .data
X:      .word 0x11111111
Y:      .word 0x22222222
        .text
        .globl main

main:   # Push return address
        addi      $sp, $sp, -4
        sw        $ra, 0($sp)

        la        $a0, X              # Address of X
        li        $a1, 127            # a = 127
        li        $a2, 0xa            # b = 0xa = 10 (decimal)
        jal       addm                # store s =  a + b in memory at address X

        la        $a0, Y              # Address of Y
        add       $a1, $v0,$zero      # a = S
        li        $a2, -3             # b = -3
        jal       addm                # store a + b in memory at address Y

        nop

        # Pop return address
        lw        $ra, 0($sp)
        addi      $sp, $sp, 4

        jr        $ra
```

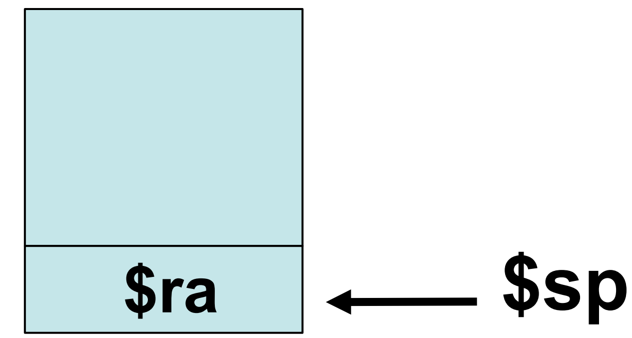
In the beginning of main we store the return address (back to the caller of main) on the stack (push).



After the PUSH

```
        .data
X:      .word 0x11111111
Y:      .word 0x22222222
        .text
        .globl main

main:   # Push return address
        addi     $sp, $sp, -4
        sw       $ra, 0($sp)

        la       $a0, X              # Address of X
        li       $a1, 127            # a = 127
        li       $a2, 0xa            # b = 0xa = 10 (decimal)
        jal      addm                # store s =  a + b in memory at address X

        la       $a0, Y              # Address of Y
        add      $a1, $v0,$zero      # a = S
        li       $a2, -3             # b = -3
        jal      addm                # store a + b in memory at address Y

        nop

        # Pop return address
        lw       $ra, 0($sp)
        addi     $sp, $sp, 4

        jr       $ra
```

In the beginning of main we store the return address (back to the caller of main) on the stack (push).

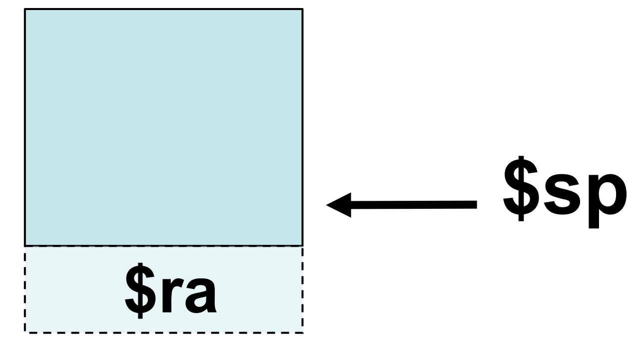


Before the POP

```
        .data
X:      .word 0x11111111
Y:      .word 0x22222222
        .text
        .globl main

main:   # Push return address
        addi    $sp, $sp, -4
        sw      $ra, 0($sp)

        la      $a0, X              # Address of X
        li      $a1, 127            # a = 127
        li      $a2, 0xa            # b = 0xa = 10 (decimal)
        jal     addm                # store s =  a + b in memory at address X

        la      $a0, Y              # Address of Y
        add     $a1, $v0,$zero      # a = S
        li      $a2, -3             # b = -3
        jal     addm                # store a + b in memory at address Y

        nop

        # Pop return address
        lw      $ra, 0($sp)
        addi    $sp, $sp, 4

        jr      $ra
```

At the end of main we restore the return address (back to the caller of main) by popping it from the stack.

In the beginning of main we store the return address (back to the caller of main) on the stack (push).



After the POP

```
        .data
X:      .word 0x11111111
Y:      .word 0x22222222
        .text
        .globl main

main:   # Push return address
        addi     $sp, $sp, -4
        sw       $ra, 0($sp)

        la       $a0, X              # Address of X
        li       $a1, 127            # a = 127
        li       $a2, 0xa            # b = 0xa = 10 (decimal)
        jal      addm                # store s =  a + b in memory at address X

        la       $a0, Y              # Address of Y
        add      $a1, $v0,$zero      # a = S
        li       $a2, -3             # b = -3
        jal      addm                # store a + b in memory at address Y

        nop

        # Pop return address
        lw       $ra, 0($sp)
        addi     $sp, $sp, 4

        jr       $ra
```

At the end of main we restore the return address (back to the caller of main) by popping it from the stack.

Whenever you write a subroutine that calls other subroutines, or calls itself (recursion), you must push the old value of $ra on the stack.

Before you do jr $ra at the end of the subroutine you must pop back the old value to $ra from the stack..

Ok, and this applies to main also since ”someone” is calling main.

For subroutines that does not call other subroutines (a.k.a leaf procedures) it is not neccessary to push and pop $ra.

The MIPS Register Convention

The first four arguments to a subroutine are passed in the registers **$a0-$a3**; subsequent arguments are passed on the stack.

Subroutine return values in ***$v0, $v1 registers***. In case of more than two return values, push return data on the stack.

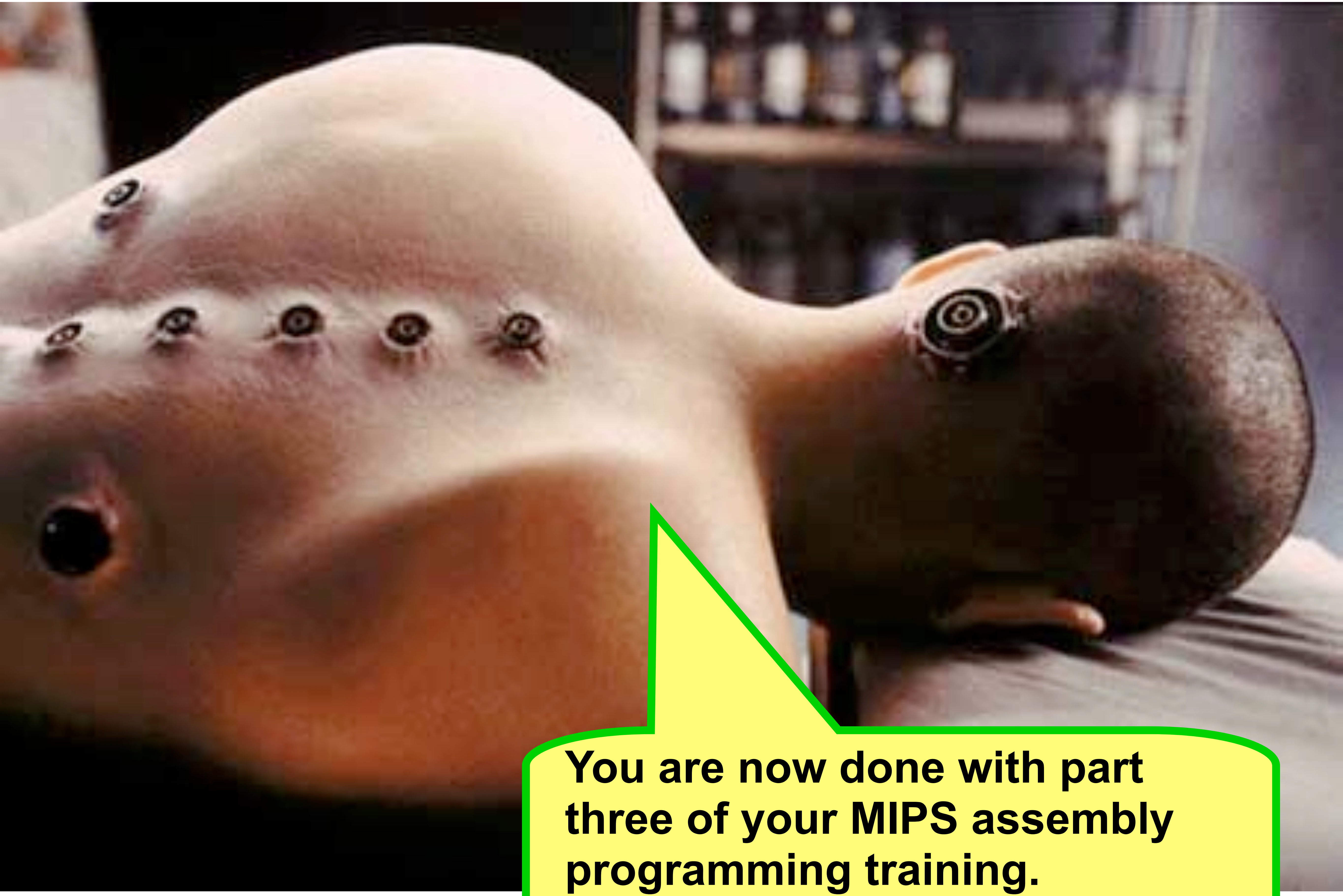
Temporary values in i ***$t-registers***.

MIPS designates 8 registers **$s0-$s7** as ***saved registers***.

It's up to the callee to save the registers if they're being used.

If you use an s-register and call a subroutien, the contents of the s-regisrers must be unchanged after the call to the subroutine.

Convention: A convention is a set of agreed, stipulated or generally accepted standards, norms, social norms or criteria, often taking the form of a custom.


You are now done with part three of your MIPS assembly programming training.