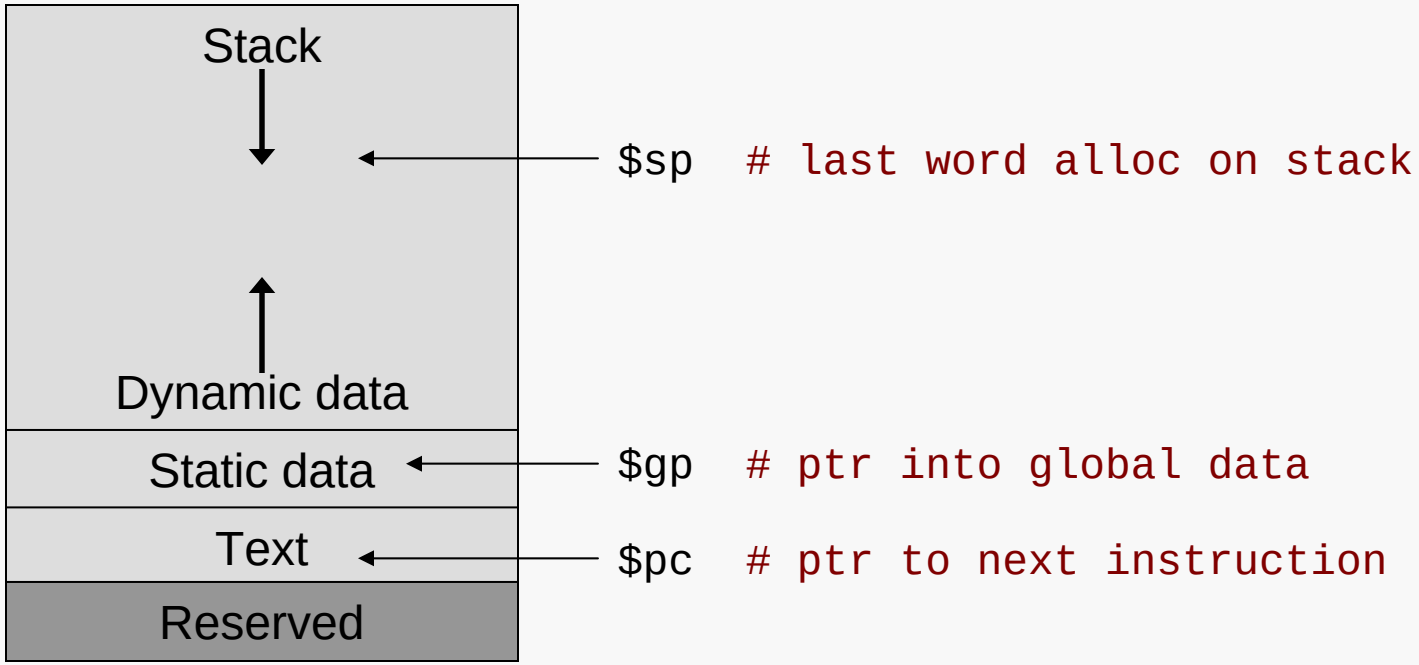
MIPS Memory Organization

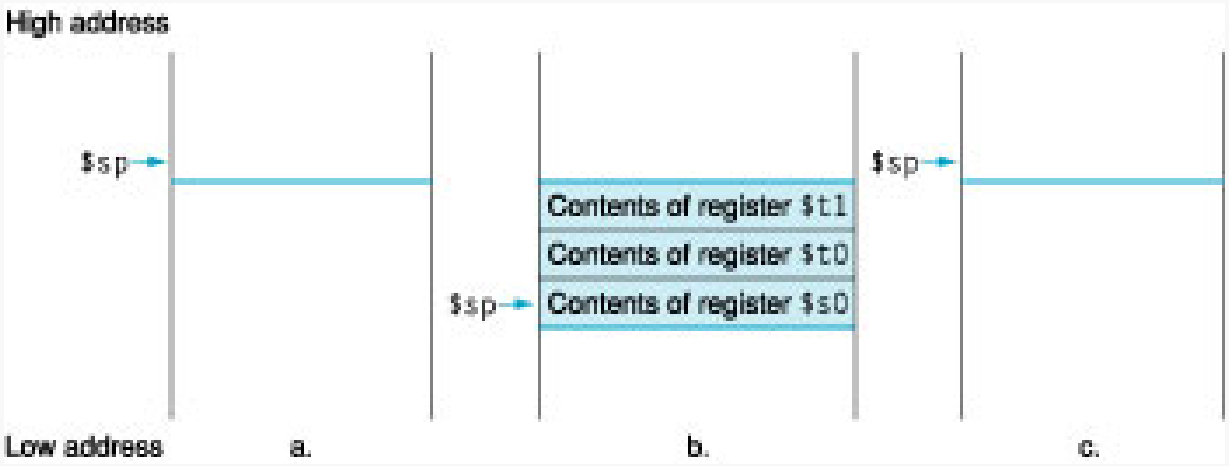
In addition to memory for static data and the program text (machine code), MIPS provides space for the run-time stack (data local to procedures, etc.) and for dynamically-allocated data:



Dynamic data is accessed via pointers held by the program being executed, with addresses returned by the memory allocator in the underlying operating system.

The System Stack

MIPS provides a special register, `$sp`, which holds the address of the most recently allocated word on a stack that user programs can employ to hold various values:



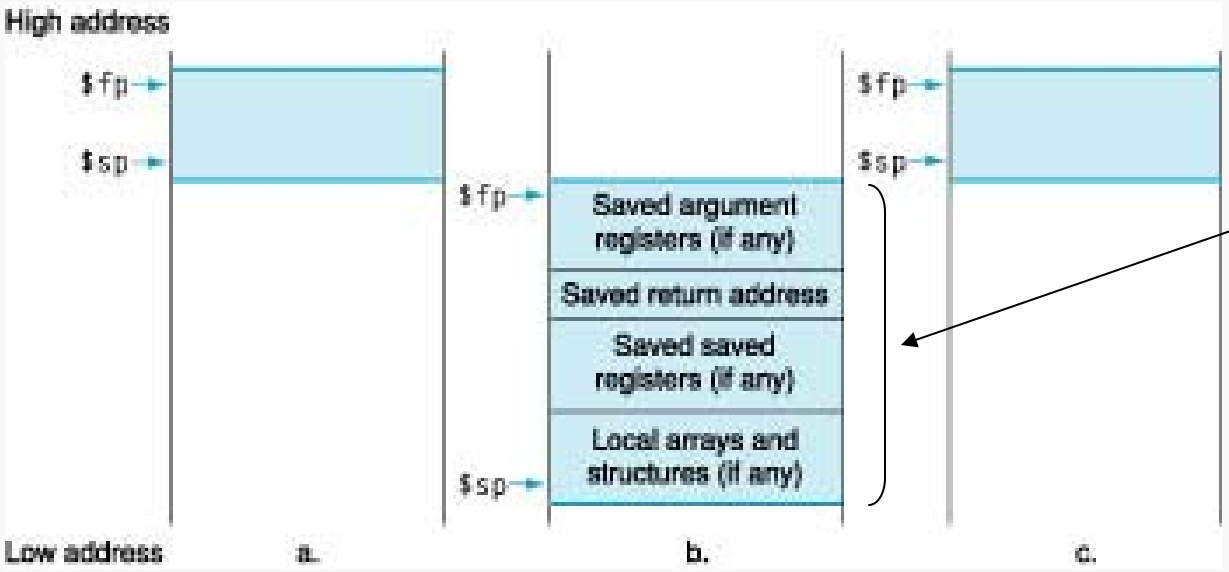
Note that this *run-time stack* is "upside-down". That is, `$sp`, decreases when a value is added to the stack and increases when a value is removed.

So, you decrement the stack pointer by 4 when pushing a new value onto the stack and increment it by 4 when popping a value off of the stack.

Using the System Stack

MIPS programs use the runtime stack to hold:

- parameters to be passed to a called procedure
- register values that need to be preserved during the execution of a called procedure and restored after the return
- saved procedure return address, if necessary
- local arrays and structures, if any



activation record or *stack frame* for called procedure

Finding the Median Value

Consider implementing a MIPS procedure to find the median value in an array of integers.

The only efficient way to do this is to partially sort the array elements until the median value is revealed.

For example, given the list of eleven values below below

```
17  43  21  19  6  34  32  25  45  29  13
```

we could sort it into the following form

```
6  13  17  19  21  25 . . .
```

and then see that the median is clearly 25.

Stack Layout

What we do not want to do is destroy the original list while finding the median.

So, we need to give our MIPS procedure a copy of the list, and the place to do that is the runtime stack.

Before calling the procedure, we will organize the stack like this:

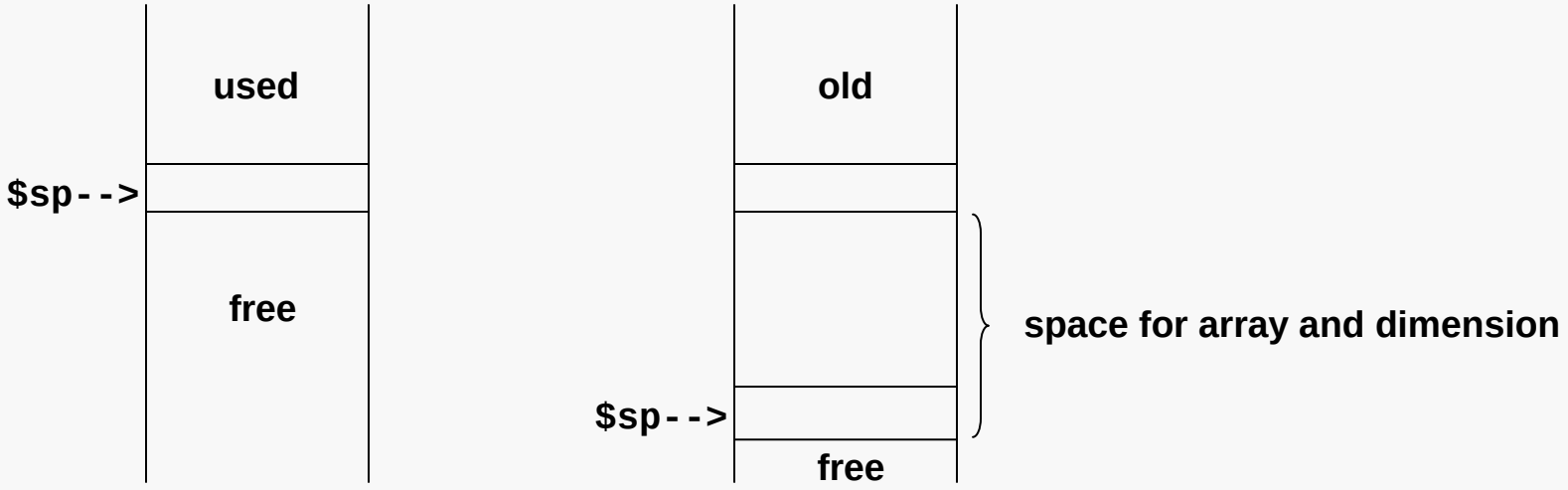
```
+---------+
| A[N-1]  |
+---------+
|  . . .  |
+---------+
| A[0]    |
+---------+
| N       | <-- $sp
+---------+
```

The procedure can then manipulate the list elements directly on the stack, and use an appropriate register to return the median value.

Creating the Stack Layout

The caller must create room on the stack to hold the necessary data:

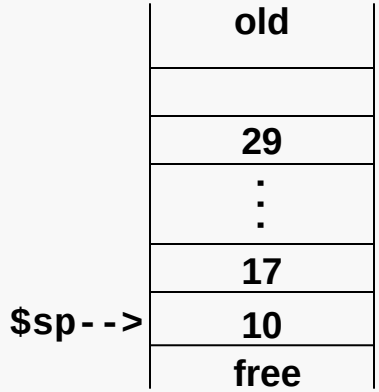
```
Size:   .word 10
List:   .word 17, 43, 21, 19, 6, 34, 32, 25, 45, 29, 13
        .text
main:
        lw    $s0, Size         # get size of list
        li    $s4, 4            # size of word
        mul   $s4, $s4, $s0     # compute size of array on stack
        addi  $s4, $s4, 4       # allow space for array dimension
        sub   $sp, $sp, $s4     # make room on stack for list and dimension
. . .
```



Creating the Stack Layout

The caller must then write the necessary data to the stack in the correct places:

```
        . . .
        sw    $s0, ($sp)        # put size onto the stack
        move  $s1, $sp          # get pointer to top of stack
        la    $s2, List         # get pointer to first list element
        li    $t0, 0            # count list elements as they are copied

loop:   beq   $t0, $s0, done
        addi  $s1, $s1, 4       # step to location for next list element
        lw    $s3, ($s2)
        sw    $s3, ($s1)        # put element onto stack
        addi  $t0, $t0, 1
        addi  $s2, $s2, 4       # step to next list element
        b     loop
done:
```



Called Procedure's View

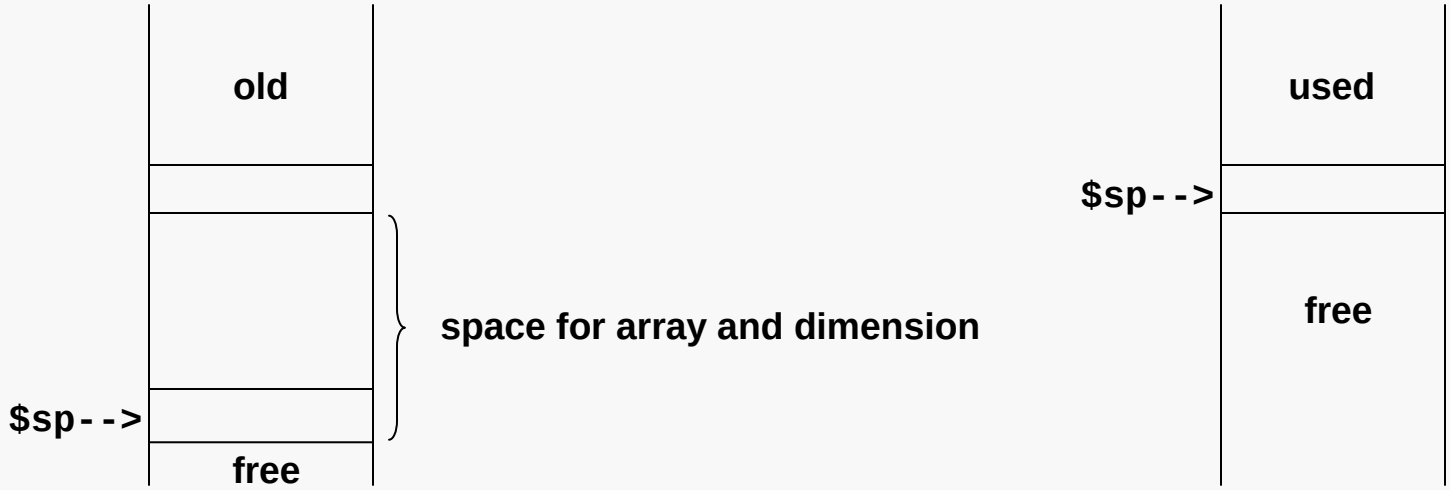
The called procedure retrieves/writes data on the stack as needed:

```
find_median:

        lw      $t3, ($sp)          # get list dimension
        addi    $t5, $sp, 4         # get address of beginning of list

        ## code to perform partial sort of array elements on the stack
        ## using any suitable algorithm

        lw      $t0, ($sp)          # get array size
        li      $t1, 2
        div     $t0, $t1            # divide it by 2
        mfhi    $t1                 # get the remainder from the divisio
        beq     $t1, $zero, even    # check whether size was even or odd

        # list has odd number of elements; set median
odd:    . . .                       # median is in middle cell of array
        b       store

        # list has even number of elements; calculate median
even:   . . .                       # median is average of two elements

store:
        swc1    $f0, ($sp)          # put median onto stack for caller
        jr      $ra                 # return to caller
```

Cleaning Up the Stack

In this case, it was the caller who put stuff onto the stack, so the caller's responsible for popping it off:

```
. . .
        jal   find_median

        l.s   $f12, ($sp)       # retrieve return value
        add   $sp, $sp, $s4     # pop stack pointer to original position
. . .
```



Local Storage on the Stack

In some cases, the called procedure needs more local storage space than the available registers can provide. For example, a procedure might need to create an array whose dimension is determined by a parameter to the procedure:

```
proc:
        . . .
        sub   $sp, $sp, $t7     # need a local array of size $t7
        move  $t0, $sp          # $t0 points to the array on the stack

        ## do stuff with the array, BUT better not mess up $t7

        add   $sp, $sp, $t7     # pop the array off the stack
        . . .
```

The Frame Pointer

The *frame pointer* register, `$fp`, is intended as a "bookmark" to keep track of where the stack pointer was when a procedure was entered:

```
proc:
        move  $fp, $sp          # $fp points to original stack top

        sub   $sp, $sp, $a0     # need a local array of size $a0
        move  $t0, $sp          # $t0 points to the array on the stack

        ## do stuff with the array; mess with $a0 all you like

        move  $sp, $fp          # restore stack pointer to original value
        jr    $ra               # right before you return
```

The $sp Contract

The implementation of a procedure must guarantee that the value of `$sp` is the same when the return is executed as it was when the procedure was entered.

Failure to do this can cause all sorts of problems…

Register Backup on the Stack

In some cases, the caller and/or the called procedure must use the stack to preserve values of $s or $t registers:

```
## Assume the caller has used the registers $t0 and $t5, and needs for those
## registers to have the same values after a procedure call:

        addi  $sp, $sp, -8   # make room on stack for two register values
        sw    $t0, 4($sp)    # back up $t0 at $sp + 4
        sw    $t5, 0($sp)    # back up $t5 at $sp

## Now push anything on the stack need to prepare for the procedure call:
        . . .
## And then make the call:
        jal   find_median
## Retrieve results from stack, if necessary, and pop all the call-related
## stuff from the stack:
         . . .
## And then retrieve the pointer values:
        lw    $t5, 0($sp)
        lw    $t0, 4($sp)
## And then pop them off the stack:
        add   $sp, $sp, 8
```

Leaf and Non-Leaf Procedures

A *leaf procedure* is one that doesn't all any other procedures.

A *non-leaf procedure* is one that does call another procedure.

Non-leaf procedures pose an additional, but simple, challenge; we make procedure calls by executing a jump-and-link instruction:

```
jal   procedure_0  # puts PC+4 into $ra for return
```

But, if procedure_0 also makes a call, say

```
jal   procedure_1  # puts PC+4 into $ra for return
```

then the original return address just got overwritten… the effect is fascinating…

Preserving the Return Address

Non-leaf procedures must back up the value of their return address before making a call to another procedure:

```
addi   $sp, $sp, -4      # make room on stack
sw     $ra, 0($sp)       # save return address
```

And they must restore the return address before they attempt to return:

```
lw     $ra, 0($sp)       # retrieve return address
addi   $sp, $sp, 4       # pop it off the stack
```

Failure to do this will almost certainly lead to a catastrophic runtime failure.

The safest way to do this is to back up the address immediately when the procedure is entered, and to restore it immediately before the return is executed. Of course, you must keep careful track of the stack pointer during all of this…