String dynamic declaration

Syntax:

.text

Create a string

v0 ← 9

a0 ← Number of bytes

Syscall

Two methods to read a string.

Method 1

$v0 ← 8

$a0 ← @ String

$a1← Number of characters to read

syscall

Method 2

Use a loop including:

$v0 ← 12 # Read a character

Syscall # $v0 contains the entered character

Procedure/Function

Procedure: a bloc of instructions that performs a specific task

Function: is a procedure that returns a value

Calling a function involves two participants:

- **Caller**: calls the function. It sets up arguments to the function, and then jumps to the function.
- **Callee**: the function itself

Procedure/Function

Function definition (by convention):

1) header
Function Name: Label
Inuput (up to 4 arguments or parameters) : in $a0-$a3

2) body
A bloc of instructions that define what the function does

3) footer
Output (up to 2 values): in $v0, $v1
Jump back to the caller

Calling a function

The caller uses the operation jal (jump-and-link) to call the function (callee).

Syntax : jal Label # Label is the name of the function

jal saves the return address (the address of the next instruction) in $ra, before jumping to the function. # PC+4 in $ra.

Calling a function

To jump back to the caller, the callee(function) has to jump to the address that was stored in $ra using the operation jr.

Syntax : jr   register

Exercise 1

Convert the following C-code into a MIPS code :

int addition(int a, int b)
{
return a+b ;
}
	void main()
	{
	int a,b ;
	scanf(''%d%d'',&a, &b) ;
	printf(''%d'',addition(a,b)) ;
	}

Exercise 2

Convert the following C-code into a MIPS code :

<pre>int SumArray(int T[], int N)
{
    int i, S=0;
    for (i = 0; i < N; i++)
        S += T[i];
    return S;
}</pre>
	<pre>void main()
	{
	    int R[10], M;
	    // Read Nb of elements M
	    // Read Array R
	    printf("%d",SumArray(R, M));
	}</pre>

Nested functions

What happens when the program (caller) calls a function that then calls another function?

Example: call a multiplication function that then calls an addition function:

7*3= 7+7+7

Let's say A (in Main) calls B (multiplication function), which calls C (MultiplicationAddition function)

A: caller

B: callee and caller at the same time

C: callee

Nested functions

Problem : jal C overwrites the return address that was saved in $ra by the earlier jal B !

Solution: The callee should save, somewhere, the return address before calling another function.

Sharing temporary registers between functions

In C languge:

Local variables in f(int a, int b) and g (int a, int b) are different.

In MIPS languge:

Temporary registers $t0-$t9 and $s0-$s7 are shared between functions (as global variables). The callee may overwrite some registers that the caller still needs them.

In a "Black box" programming approach: different functions may be written by different programmers or companies. A function should be able to interface with any client.

Sharing temporary registers between functions

How to manage access to these registers?

Case 1: the caller must save registers that it needs before making a function call, and restore them after.

Problem: the caller does not know what registers will be used by the callee. Many registers may be unnecessarily saved.

Sharing temporary registers between functions

How to manage access to these registers?

Case2: the callee must save and restore any registers it might overwrite.

Problem: the callee does not know what registers are important to the caller. Many registers may be unnecessarily saved.

Sharing temporary registers between functions

How to manage access to these registers?

Solution: share the backup and restore task between the caller and the callee

Two types of registers (by convention):

Caller-saved (12 registers): $v0-$v1, $t0-$t9

Callee-saved (12 registers): $a0-$a3, $s0-$s7

The caller saves and restores any caller-saved registers that it cares about.

The callee saves and restores any callee-saved registers that it uses.

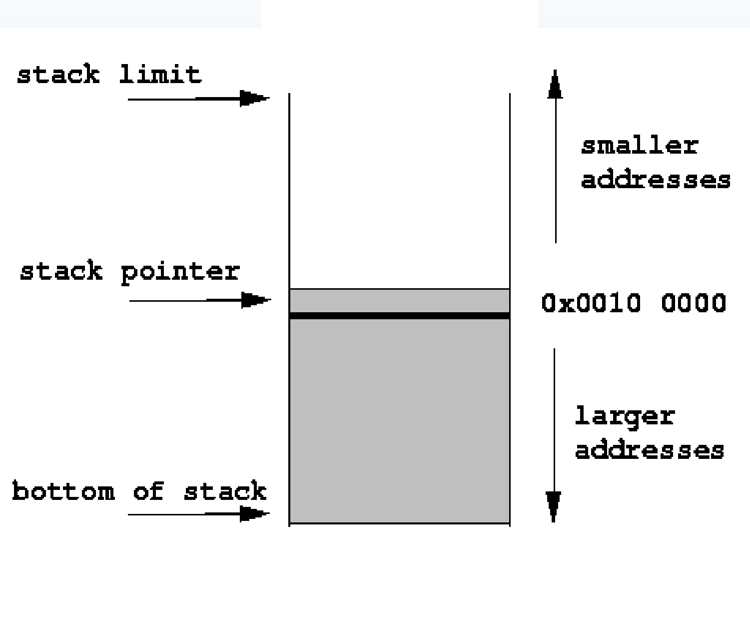
Where to save registers' contents?

A stack is a part of the RAM.

Elements are inserted (pushed) and removed (popped) according to the last-in first-out (LIFO) principle.

In MIPS, the stack grows downward in terms of memory addresses.

The address of the top element is stored in $sp (stack-pointer register).



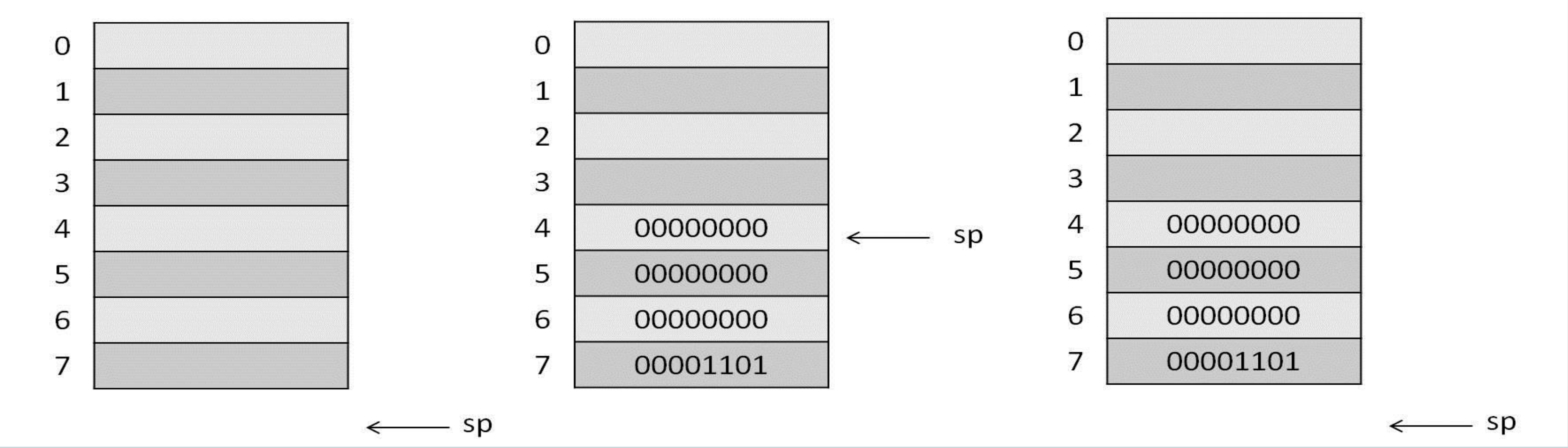
Stack operations

Push:
$sp ← $sp -4 # Allocate memory. Decrement stack pointer by 4
sw $t1, ($sp) # Store

Pop:
lw $t1, ($sp) # Load
$sp ← $sp +4 # Free memory. Increment stack pointer by 4

Example

Consider the following stack. Add, in binary, the integer 13 to the stack and restore it.



Exercise 1

Write a program that asks user to enter an integer then does the following:

- Push a to the stack.
- Print “Pushed”.
- Pop a from the stack.
- Print “Popped”.

Exercise 2

Write a MIPS program that includes the following functions:

- Create: an integer array of size N.
- Fill: an integer array of size N.
- Print: an integer array of size N.
- Replace: an integer by 0.

Test the above functions.