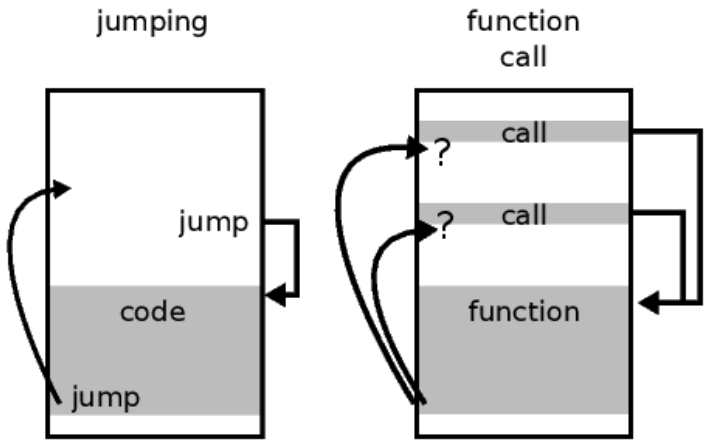
Computer Architecture

Exercise 6
(functions)

Peter Stallinga
Universidade do Algarve 2024-2025

Functions are temporary interruptions from the flow of the program and execute another part of the code. These functions can be called from any part of the program, so we have to remember where the program was interrupted.
Compare to how we used unconditional (`j`) and conditional jumps (branch, `bne`, etc) until now (left). The exact place to jump (back) to is known.



Compare that to functions, or subroutines (right). After the function is finished we need to go back to where the program was interrupted. For that we have to remember the address of the instruction following the one that was calling the function. The instruction for that is `jal` (jump-and-link),

```
jal address
```

It saves the program counter + 4 (meaning the address of the *next* instruction) in the return-address register `$ra`, and then sets the program counter to the address.

`$pc+4` → `$ra`
`address` → `$pc`

At the end of the function we can then simply use the jump-register (`jr`) instruction:

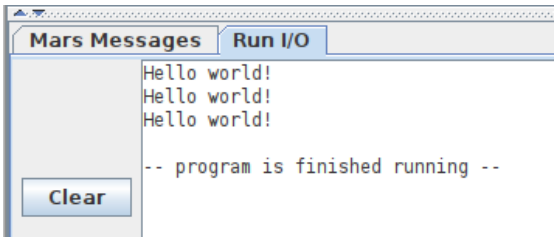
```
jr $ra
```

which is equivalent to

`$ra` → `$pc`

This is equivalent to the `return()` instruction in C.

Exercise 1: Write a function that prints the text “Hello world!”. In the main part of the program call the function three times (do not use a for-loop).



Note that the 'main' code now essentially needs to be finished with a syscall 10 part, otherwise it would start executing the code of the function …

If we want the function subroutine to return a value, we can place that conventionally in the `$v0` register (and `$v1` if we want two values). If we want to pass arguments to the function we can place them in the `$a` registers.

Exercise 2: Write a function that calculates the square of the argument and returns it. In the main function print the table of squares from 1 to 10.

Macros are not functions, although they look very similar to functions in languages like C. In reality a macro is a description for the compiler/assembler to do a pre-translation before the real compilation. When the label of the macro is encountered, the compiler simply substitutes it with the macro. Take a look at this example:

```
.macro print (%ttt)
  li $v0, 4
  la $a0, %ttt
  syscall
.end_macro
```

After the definition of the macro, every time the label `print` is encountered our assembler substitutes it with the correct MIPS code. Note also that a macro can have parameters, as in this example, with `%ttt`. If we now write in our MIPS program

```
  print hellow
```

it will be substituted with

```
  li $v0, 4
  la $a0, hellow
  syscall
```

and then the program is assembled.

Exercise 3: Write a macro that implements `boe`, branch on even, with a register as a parameter. So

```
boe $t1, jumpaddress
```

will jump to `jumpaddress` when `$t1` is even.
Test your macro by changing the program printing squares to only print squares of odd numbers,

The relevant (new) instructions for today are:

`jal`	Jump-and-link
`jr`	Jump register
`.macro ... .end_macro`	macro