Machine Architecture - Lecture 7

```
# Get n from user and save
                li              $v0, 5    # read integer
                syscall
                move            $t0, $v0  # syscall result

# Initialize registers
                li              $t1, 0 # initialize counter i
                li              $t2, 0 # initialize sum

                # Main loop body
loop:           addi            $t1, $t1, 1       # i = i + 1
                add             $t2, $t2, $t1     # sum = sum + i
                beq             $t0, $t1, exit    # break from loop
                j               loop

exit:           # Print sum
                li              $v0, 1     # print_string
                move            $a0, $t2
                syscall

                # Exit
                li              $v0, 10    # exit
                syscall
```

Ioannis Ivrissimtzis

ioannis.ivrissimtzis@durham.ac.uk

MIPS – programming

Slide material acknowledgements: Magnus Bordewich

Functions

Functions are pieces of code which can be accessed from other parts of the program.

Using functions makes code more modular and readable.

MIPS assembly functions have inputs, called arguments, and an output called return value.

Need an agreement on how to:

- call and return from a function
- access the input arguments and the return value

Call and return

```
                # The caller function "main"
0x00400200      main:   jal     simple
0x00400204              ...

                # The callee function "simple"
0x00401020      simple: jr      $ra
```

The function **main** calls the function **simple** using the **jal** instruction.

jal simple (**j**ump **a**nd **l**ink) jumps to the address **0x00401020** (as **j** would do) but also stores in register **$ra** the address where the program should return after **simple** has been executed (here **0x00400204**).

jr $ra (**j**ump **r**egister) jumps to the address stored in an register (here **$ra**). Notice, that **jr** is an R-type not a J-type instruction.

Arguments and return value

```
main:    li   $a0, 10              # argument 0 gets the value 10
         li   $a1, 5               # argument 1 gets the value 5
         li   $a2, 20              # argument 2 has value 20
         li   $a3, 10              # argument 3 has value 10
         jal  diffofsums           # call the function
         move $s0, $v0             # put return value to $s0
         ...
diffofsums:
         add  $t0, $a0, $a1        # sum of the first two arguments
         add  $t1, $a2, $a3        # sum of the other two arguments
         sub  $t2, $t0, $t1        # difference of the two sums
         move $v0, $t2             # put return value at $v0
         jr   $ra                  # return to caller
```

`move $s0, $v0` is another pseudo-instruction like **`li`**. It copies the value of a register into another register. It is implemented as:

`add $s0, $v0, $0`

Arguments and return value

According to MIPS conventions on the behaviour of caller and callee:

the caller places the arguments into the registers **`$a0`** - **`$a3`**

the return value is placed into the registers **`$v0`** - **`$v1`**

the saved registers **`$s0`** - **`$s7`** are not modified by the callee

This convention can be quite restrictive, especially if the callee is going to call another function (or even call itself recursively).

Instead of relying to this convention, the callee can first save all important register in a stack (a data structure covered at the ADS module) and restore them before returning to the caller.

store value and after execution. before execution stack pointer must be the same

Loops

Exercise: Compute the n-th triangular number.

The program should take the input n and output T(n) where:

$$T(n) = 1 + 2 + ... + n$$

Loops

The code fragment shows register initialisation and the main loop.

T(10)

```
        # Initialize registers
        li      $t0, 10          # load the value of N
        li      $t1, 0           # initialize the counter (i)
        li      $t2, 0           # initialize sum

        # Main loop body
loop:   addi    $t1, $t1, 1      # i = i + 1 (increment the counter)
        add     $t2, $t2, $t1    # sum = sum + i
        beq     $t0, $t1, exit   # if i = N, break from the loop
        j       loop

exit:   ...
        ...
```

Loops

The main loop is implemented through an unconditional jump instruction **j** and a branch on equal instruction **beq**, which branches out of the loop when the values in the registers **$t0** and **$t1** become equal.

```
        # Main loop body
loop:   addi    $t1, $t1, 1       # i = i + 1 (increment the counter)
        add     $t2, $t2, $t1     # sum = sum + i
        beq     $t0, $t1, exit    # if i = N, break from the loop
        j       loop

exit:   ...
        ...
```

Input / Output

Can we get the value of n from the user, through the keyboard, instead of hard encoding it?

```
# Get n from user and save
li        $v0, 5     # load the syscall code 5 to $v0
syscall              # read integer (syscall code is 5)
move      $t0, $v0   # syscall result (returned in $v0) moves to $t0
```

The syscall instruction (system call) suspends the execution of the program to provide an operating-system-like service, such as input, output, termination).

Input

```
# Get n from user and save
li        $v0, 5    # read integer (syscall code is 5)
syscall
move      $t0, $v0  # syscall result (returned in $v0) move to $t0
```

The type of the syscall service is specified by a code, which should be stored in **$v0**.

The code 5 used here corresponds to reading an integer from keyboard.

Examples of syscall services

service	syscall code	arguments	result
print integer	1	$a0 = integer	-
print string	4	$a0 = string	-
read integer	5	-	integer (in $v0)
exit	10	-	-

Output and exit

After exiting the main loop, we print the output and stop the execution.

```
exit:   # Print sum
        li      $v0, 1          # print_string syscall code = 1
        move    $a0, $t2
        syscall

        # Exit
        li      $v0, 10         # exit
        syscall
```

It is important, always, to declare the end of the program.

Otherwise, the computer will fetch the word stored immediately after the last instruction and try to execute it with unpredictable behaviour.

Putting it all together … so far

```
# Get n from user and save
        li      $v0, 5      # read integer (syscall code is 5)
        syscall
        move    $t0, $v0    # syscall result (returned in $v0) move to $t0

# Initialize registers
        li      $t1, 0              # initialize the counter (i)
        li      $t2, 0              # initialize sum.

        # Main loop body
loop:   addi    $t1, $t1, 1         # i = i + 1 (increment the counter)
        add     $t2, $t2, $t1       # sum = sum + i
        beq     $t0, $t1, exit      # if i = N, break from the loop
        j       loop

exit:   # Print sum
        li      $v0, 1          # print_string syscall code = 1
        move    $a0, $t2
        syscall

        # Exit
        li      $v0, 10         # exit
        syscall
```

N = int(input())
sum, counter = 0, 0
while i != N:
    i = i + 1
    sum = sum + i
print(sum)

Text output

To make the problem more user friendly, we want to add some text output describing the format of the requested input and also what the output is.

See the program triangularNumber.asm uploaded on DUO.

Where is the .data segment of the program stored?
Next … the MIPS memory map.