Computer Architecture (Lab). Week 1

Vladislav, Artem, Hamza, Manuel

Innopolis University

vl.ostankovich@innopolis.ru
a.burmyakov@innopolis.ru
h.salem@innopolis.university
m.rodriguez.osuna@innopolis.university

September 3, 2020

Topic of the lab

- Journey from C to Assembler

Contents

Introduction

- To command a computer, you must understand its language
  - **Instructions:** words in a computer's language
  - **Instruction set:** the vocabulary of a computer's language

- Instructions indicate the operation to perform and the operands to use
  - **Assembly language**: human-readable format of instructions
  - **Machine language:** computer-readable format (1's and 0's)

Because of the close relationship between machine and assembly languages, each different machine architecture usually has its own assembly language (in fact, each architecture may have several), and each is unique.

Introduction

- **MIPS** architecture
  - Developed by John Hennessy and colleagues at Stanford in the 1980's
  - Used in many commercial systems (Silicon Graphics, Nintendo, Cisco)
  - **M**icroprocessor without **I**nterlocked **P**ipeline **S**tages is a reduced instruction set computer (RISC) instruction set architecture (ISA) developed by MIPS Computer Systems (now MIPS Technologies).

MIPS

- Five most iconic devices that use MIPS CPUs:





MIPS
by Imagination

MIPS (cont.)

MIPS is currently used in many embedded systems

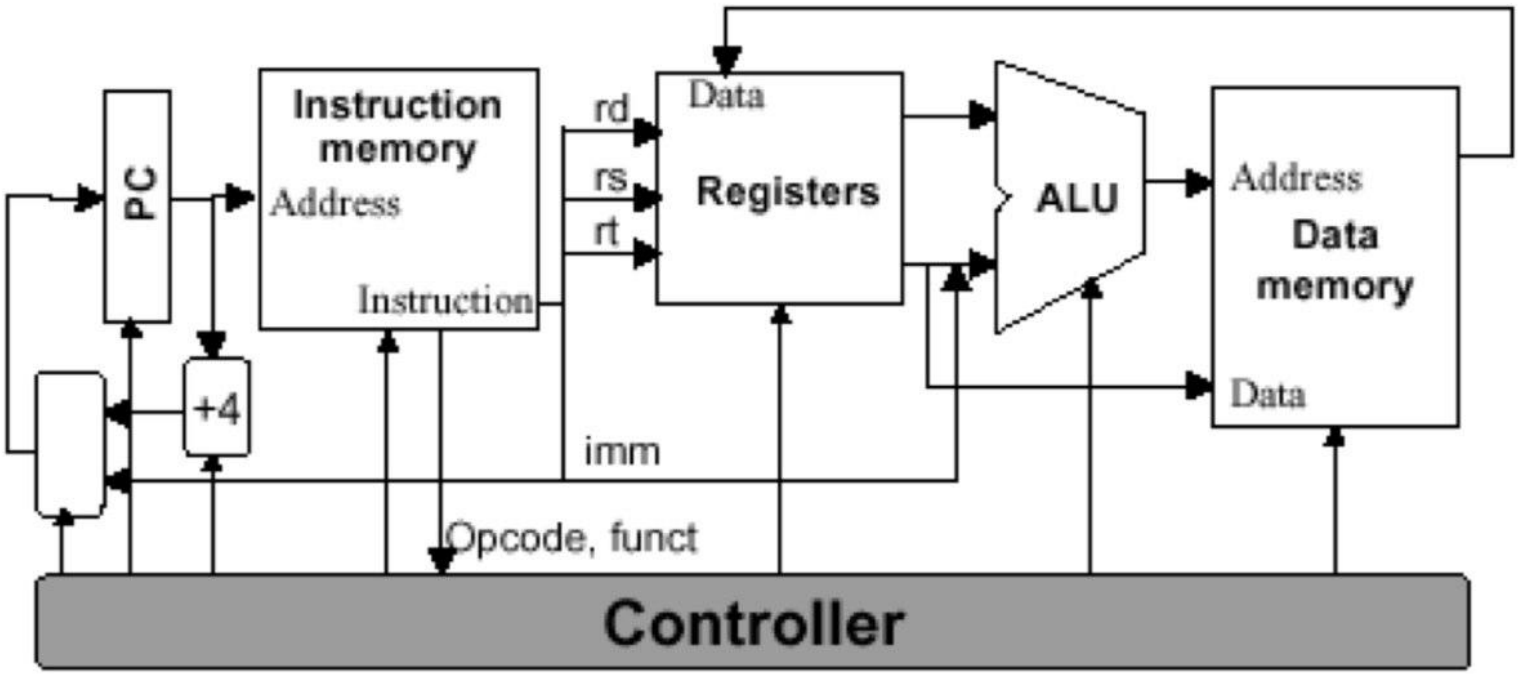
- Android supports MIPS hardware platforms
- DVD, Digital TV, Set-Top Box
- Nintendo 64, Sony Playstation and Playstation 2
- Cisco routers

Check out the MIPS web page for more information: www.mips.com

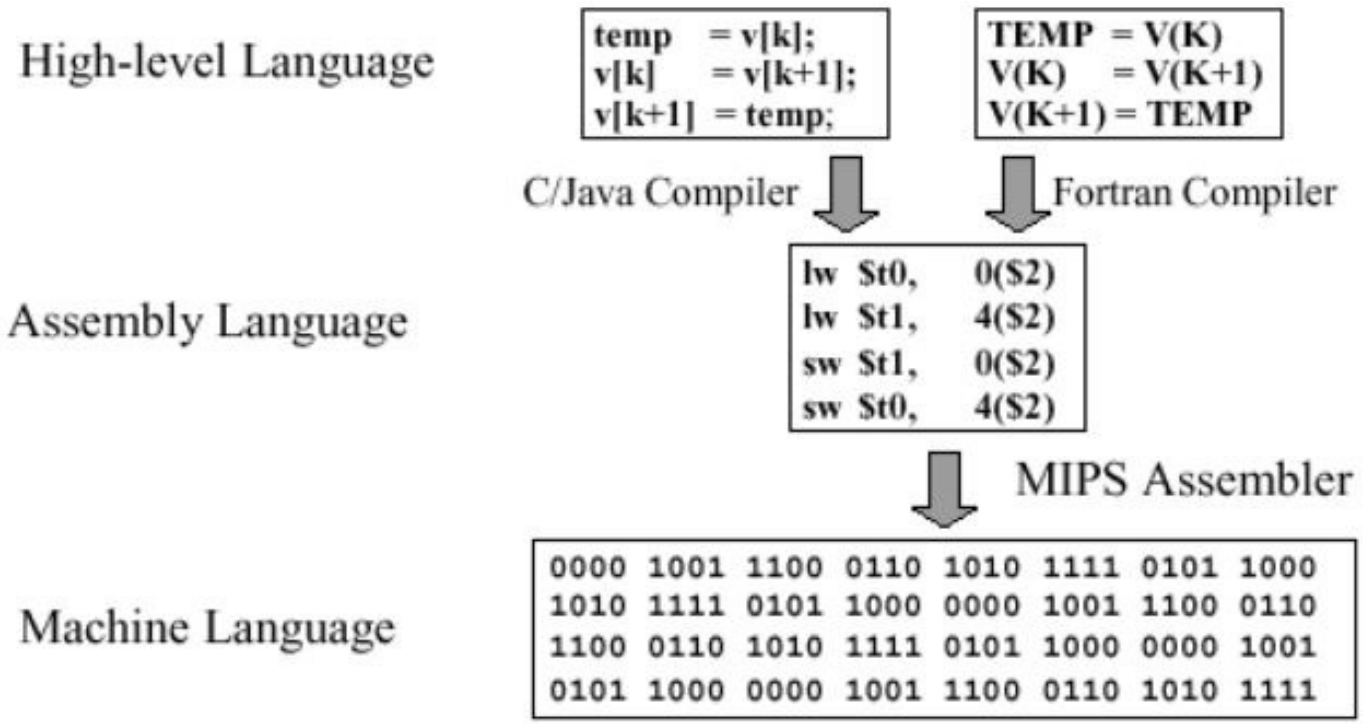
Translation

- The assembly language level differs in a significant respect from the Microarchitecture, Instruction set architecture, and Operating system machine levels – it is implemented by **translation** rather than by **interpretation**.
- Programs that convert a user's program written in some language to another language are called **translators**.
- The language in which the original program is written is called the **source** language.
- The language to which it is converted is called the **target** language.

Why translate?



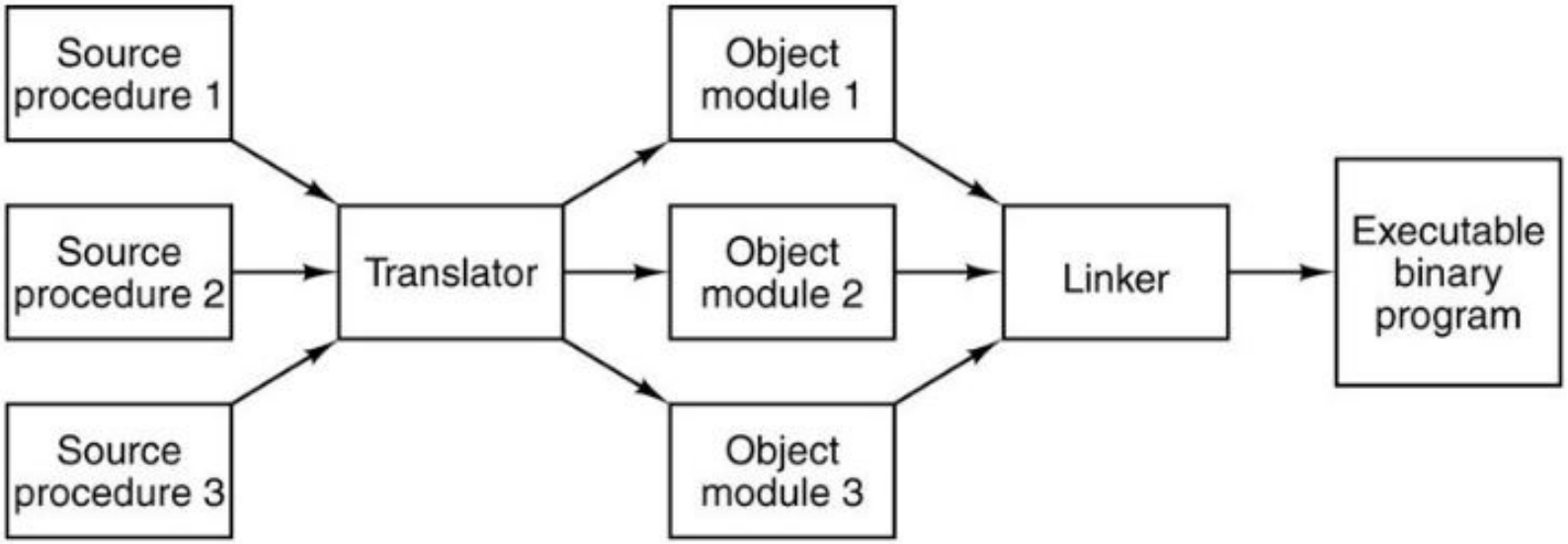
Example of computing language translation



What is translation?

- In translation, the original program in the source language is not directly executed. Instead, it is converted to an equivalent program called an **object program** or **executable binary program** whose execution is carried out only after the translation has been completed.
- In translation, there are two distinct steps:
  - Generation of an equivalent program in the target language.
  - Execution of the newly generated program.
- **In translation**, these two steps do not occur simultaneously. The second step does not begin until the first has been completed.
- **In interpretation**, there is only one step: executing the original source program.

Translator


Source procedure 1
Source procedure 2
Source procedure 3
Translator
Object module 1
Object module 2
Object module 3
Linker
Executable binary program

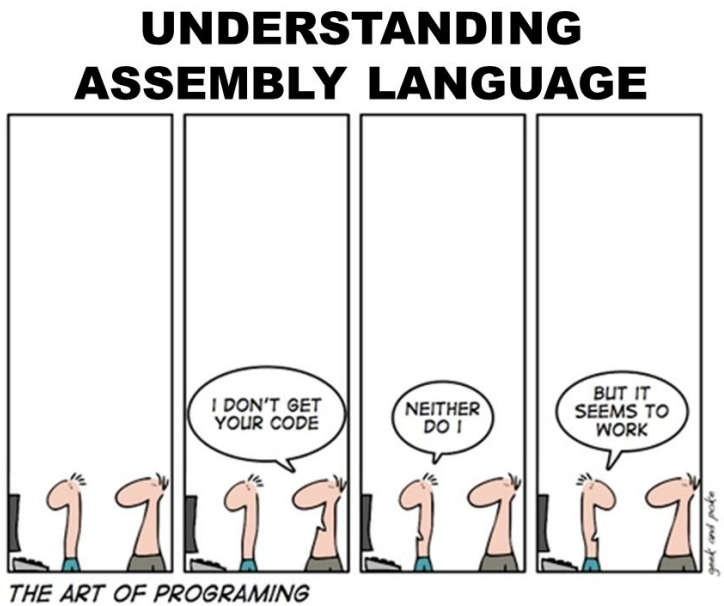
Types of Translator

- Translators can be roughly divided into two groups, depending on the relation between the source language and the target language.
- When the source language is essentially a symbolic representation for a numerical machine language, the translator is called **an assembler** and the source language is called **an assembly language**.
- When the source language is a high-level language such as Java or C and the target language is either a numerical machine language or a symbolic representation for one, the **translator** is called **a compiler**.

What is an Assembly Language?

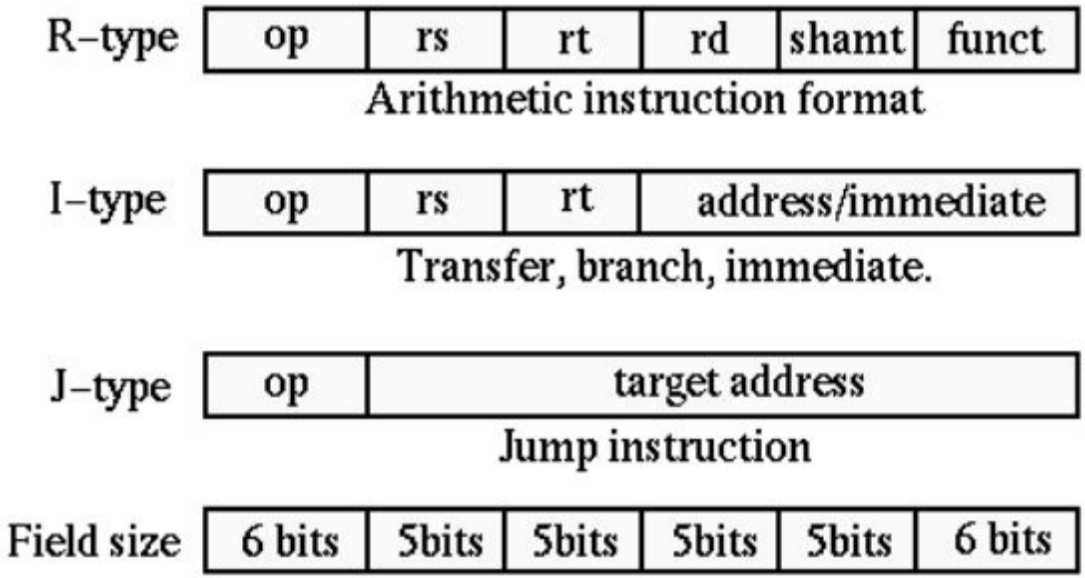
- A pure assembly language is a language in which each statement produces exactly one machine instruction.

- There is a one-to-one correspondence between machine instructions and statements in the assembly program.



Machine Language Instruction Format

- In machine language, all instructions have the same length.
- In MIPS instructions are 32 bits long.
- MIPS instructions have three different formats:



Assembly versus Machine Language

- Assembly language is easier to use than machine language.
- The assembly language programmer need only remember the symbolic names because the assembler translates them to the machine instructions.
- The assembly language programmer can give symbolic names to memory locations and have the assembler to worry about supplying the correct numerical values.
- The machine language programmer must always work with the numerical values of the addresses. As a consequence, no one programs in machine language today, although people did so decades ago, before assemblers had been invented.

Assembly versus High-level language

- The assembly programmer has access to all the features and instructions available on the target machine. The high-level language programmer does not.

- Everything that can be done in machine language can be done in assembly language, but many instructions, registers, and similar features are not available for the high-level language programmer to use.

- An assembly language program can only run on one family of machines, whereas a program written in a high-level language can potentially run on many machines. For some applications, this ability to move software from one machine to another is of great practical importance.

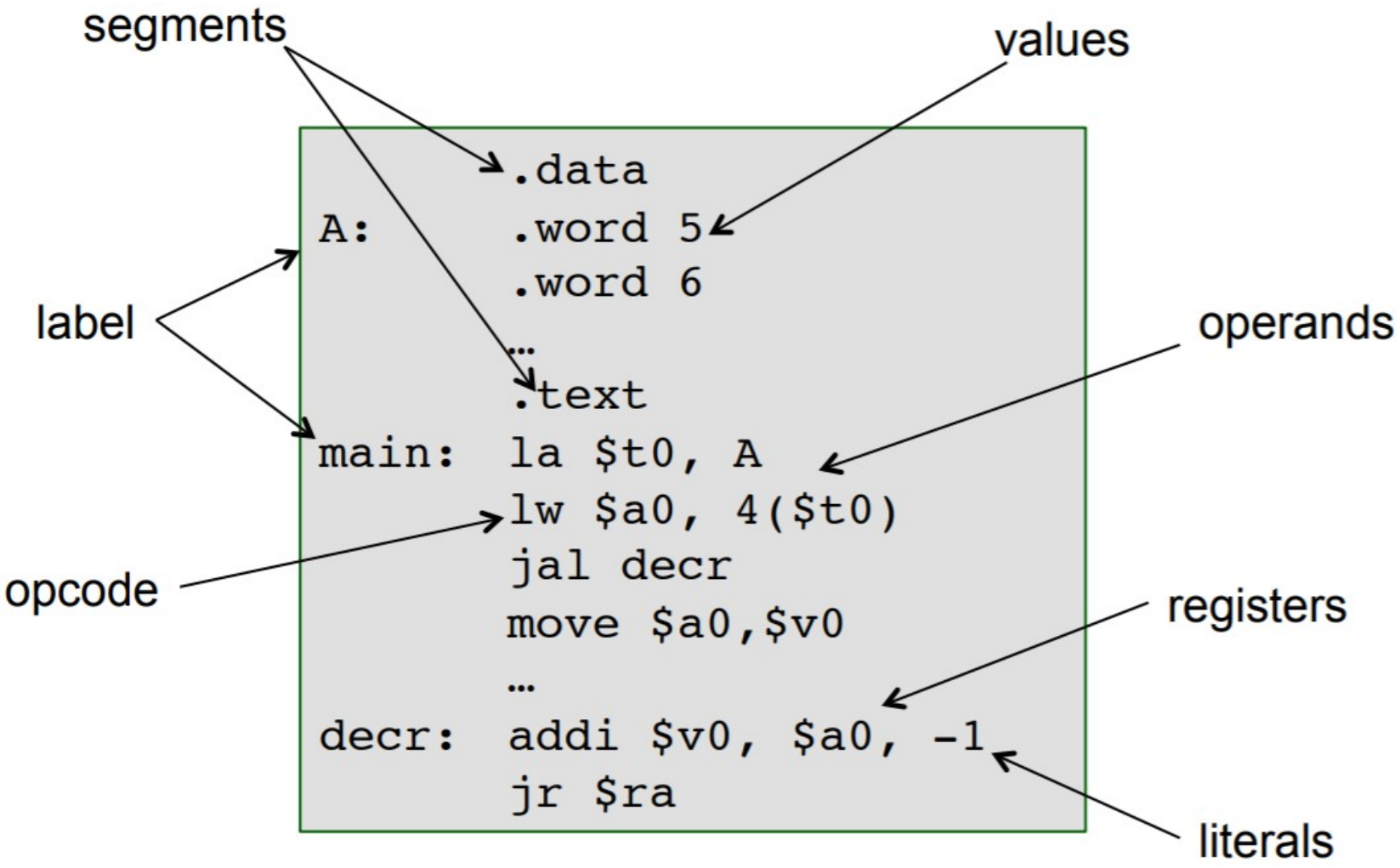
Why Assembly Language?

- Writing a program in assembly language takes much longer than writing the same program in a high-level language.
- It also takes much longer to debug and is much harder to maintain.
- However, there are two reasons for using assembly language: **performance** and **access to the machine**.

Performance versus Machine Access

- Performance:
  - An expert assembly language programmer can often produce code that is much smaller and much faster than a high-level language programmer can.
  - For some applications, speed and size are critical. For example, smart cards, embedded applications, device drivers etc.
- Access to the machine:
  - Some procedures need complete access to the hardware, something usually impossible in high-level languages.
  - For example, the low-level interrupt and trap handlers in an operating system, and the device controllers in many embedded realtime systems fall into this category.

MIPS Assembly Language



MIPS Registers

Name	Register Number	Usage
$zero	0	constant 0 (**hardwired**)
$at	1	**reserved** for assembler
$v0 - $v1	2-3	returned values
$a0 - $a3	4-7	arguments
$t0 - $t7	8-15	temporaries
$s0 - $s7	16-23	saved values
$t8 - $t9	24-25	temporaries
$gp	28	global pointer
$sp	29	stack pointer
$fp	30	frame pointer
$ra	31	return address

MIPS Instructions

MIPS Instructions	Name	Format
addu	`addu`	R
subtract	`subu`	R
add immediate	`addiu`	I
load word	`lw`	I
store word	`sw`	I
load byte	`lb`	I
store byte	`sb`	I
load upper immediate	`lui`	I
branch on equal	`beq`	I
branch on not equal	`bne`	I
set less than	`slt`	R
set less than immediate	`slti`	I
jump	`j`	J
jump register	`jr`	R
jump and link	`jal`	J

Hello World in C

```c
#include <stdio.h>
int main() {
      printf("hello world\n");
      return 0;
}
```

Hello World in MIPS Assembler Language

```
# Hello, World!
        .data     ## Data declaration section
## String to be printed:
out_string:    .asciiz   "\nHello, World!\n"

        .text     ## Assembly language instructions go in text segment
main:             ## Start of code section
        li     $v0, 4           # system call code for printing string = 4
        la     $a0, out_string  # load address of string to be printed into $a0
        syscall                 # call operating system to perform operation
                                #   specified in $v0
                                # syscall takes its arguments from $a0, $a1, ...

        li     $v0, 10          # terminate program
        syscall
```

Other Programs

```
1
2  int main(){
3    int a = 9;
4    int b = 7;
5    int c = 4;
6    c = a+b;
7    return 0;
8  }
9
```

```
1  main:
2          addiu   $sp,$sp,-32
3          sw      $fp,28($sp)
4          move    $fp,$sp
5          li      $2,9                    # 0x9
6          sw      $2,8($fp)
7          li      $2,7                    # 0x7
8          sw      $2,12($fp)
9          li      $2,4                    # 0x4
10         sw      $2,16($fp)
11         lw      $3,8($fp)
12         lw      $2,12($fp)
13         nop
14         addu    $2,$3,$2
15         sw      $2,16($fp)
16         move    $2,$0
17         move    $sp,$fp
18         lw      $fp,28($sp)
19         addiu   $sp,$sp,32
20         j       $31
21         nop
```

https://godbolt.org/

Acknowledgements

- This lab was created and maintained by Vitaly Romanov, Aidar Gabdullin, Munir Makhmutov, Ruzilya Mirgalimova, Muhammad Fahim, Vladislav Ostankovich, Alena Yuryeva, Artem Burmyakov, Hamza Salem and Manuel Rodriguez