ARCOS Group

uc3m | Universidad **Carlos III** de Madrid

L3: Fundamentals of assembler programming
Computer Structure

Bachelor in Computer Science and Engineering

Bachelor in Applied Mathematics and Computing

Dual Bachelor in Computer Science and Engineering and Business Administration

Contents

1. **Basic concepts on assembly programming**
    1. Motivations and goals
    2. Introduction to RISC-V32
2. RISC-V32 assembly language, memory model and data representation
3. Instruction formats and addressing modes
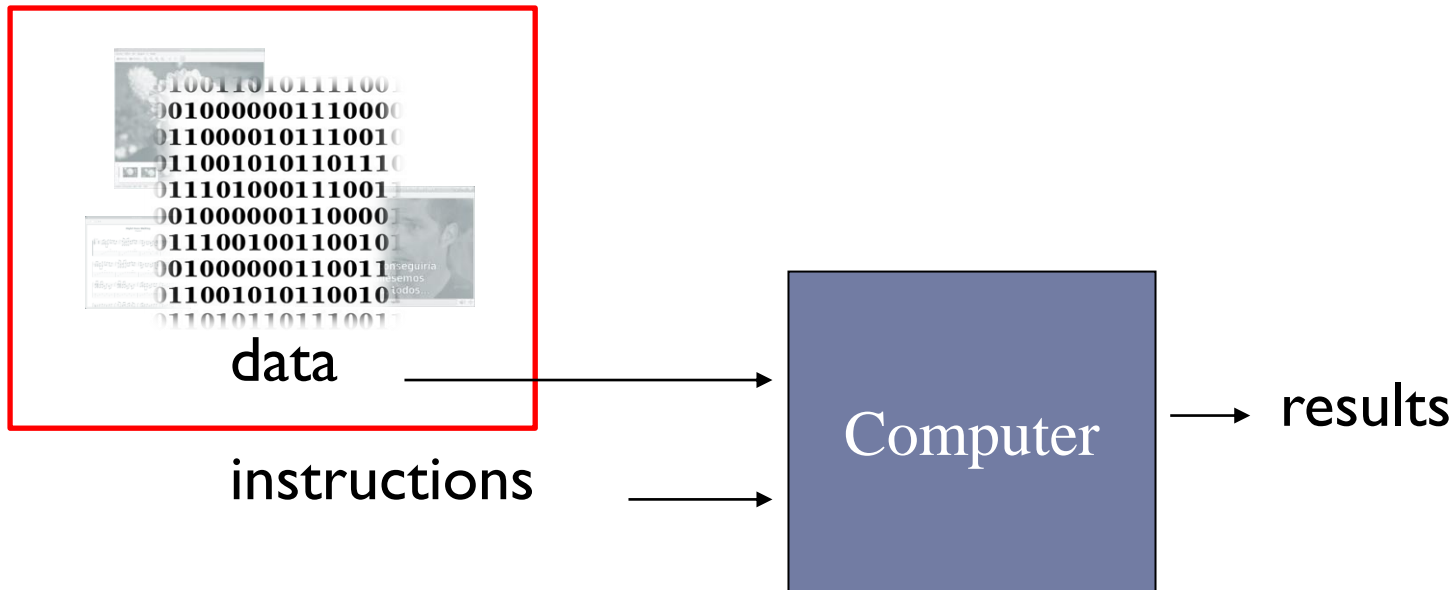4. Procedure calls and stack convention

Types of information: instructions and data

▶ Data representation...

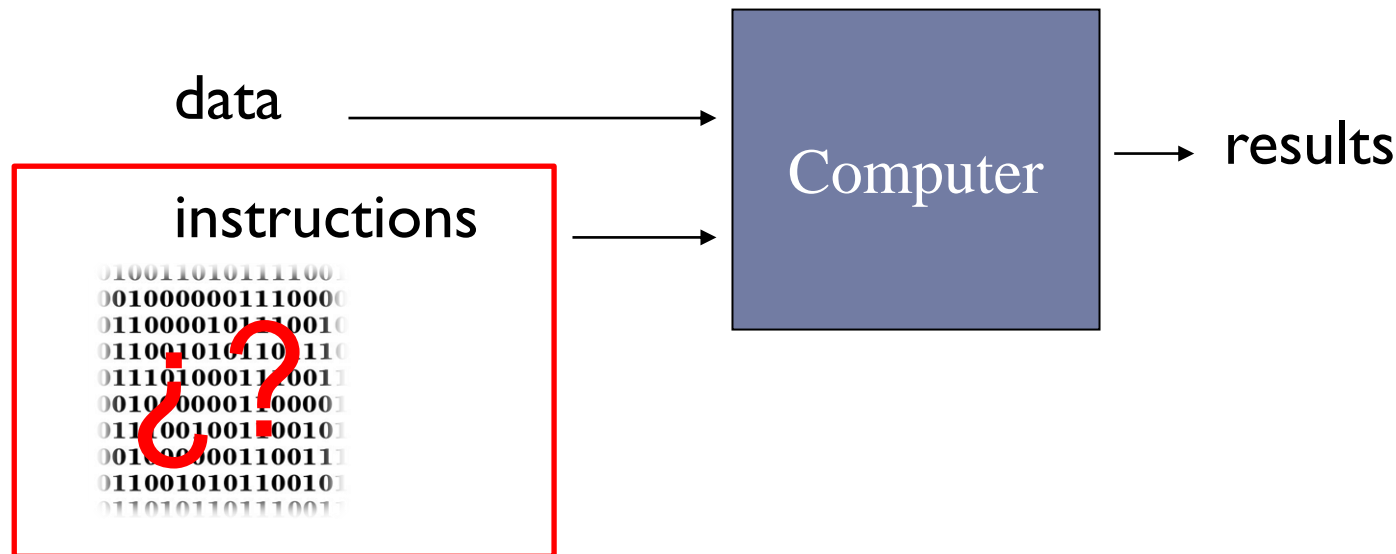

data

instructions

Computer

→ results

Félix García-Carballeira,   Alejandro Calderón Mateos
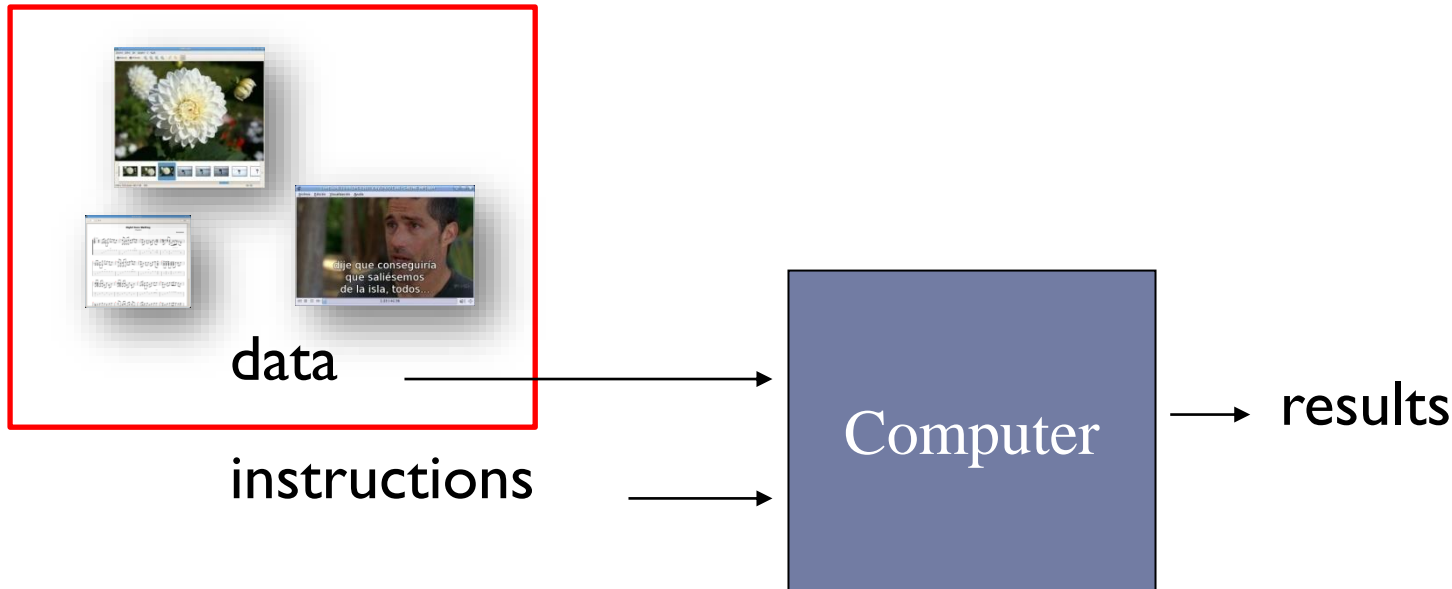
Types of information: instructions and data

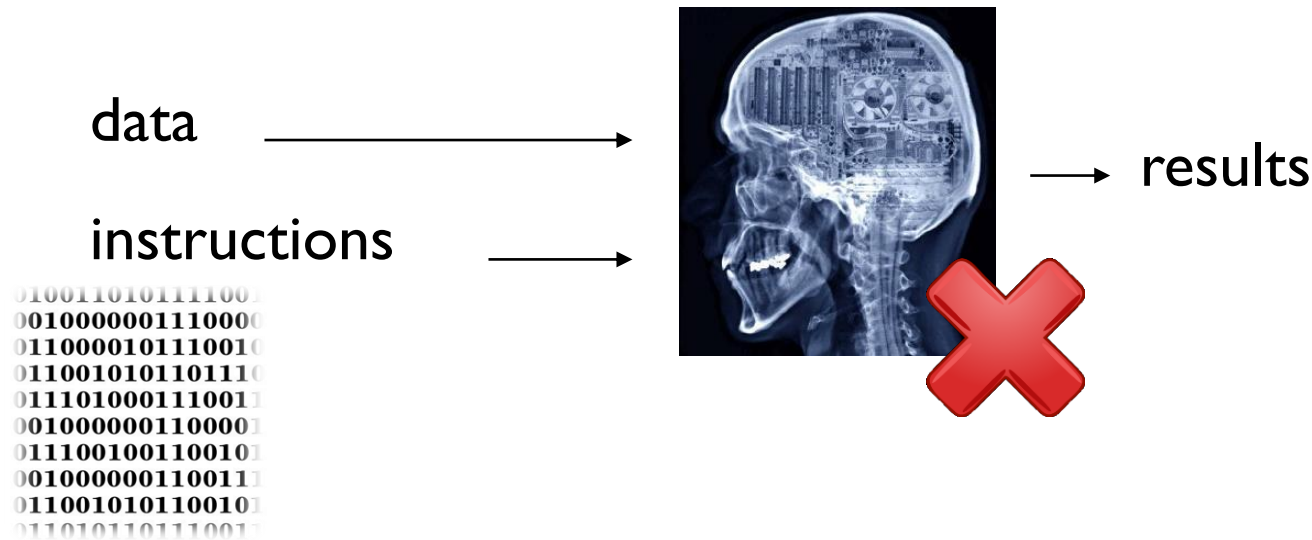▸ **Binary data** representation.

data →

instructions →

Computer → results

Types of information: instructions and data

▸ **What about the instructions?**

   ▸ Machine instructions, properties and format

data → **Computer** → results

instructions →

Félix García-Carballeira, Alejandro Calderón Mateos

Machine instructions

▸ There are not complex instructions…

data →

instructions →

results

https://encrypted-tbn0.gstatic.com/images?q=tbn:ANd9GcQYvoToQOCuBYYDzqvn--YxoWFvJ5u4zqteF-Q47AYqxh2jQC5j

ARCOS @ UC3M

Félix García-Carballeira, Alejandro Calderón Mateos

Machine instructions

▸ … but very simple tasks…

data ⟶

instructions ⟶

 ⟶ results

```
0100110101111001
0010000001110000
0110000101110010
0110010101101110
0111010001110011
0010000001100001
0111100100110010
0010000001100111
0110010101100010
0110101101110011
```

https://miro.medium.com/fit/c/1360/1360/1*cf7_cdoVAW3-AOL6XxG3Xw.jpeg

ARCOS @ UC3M

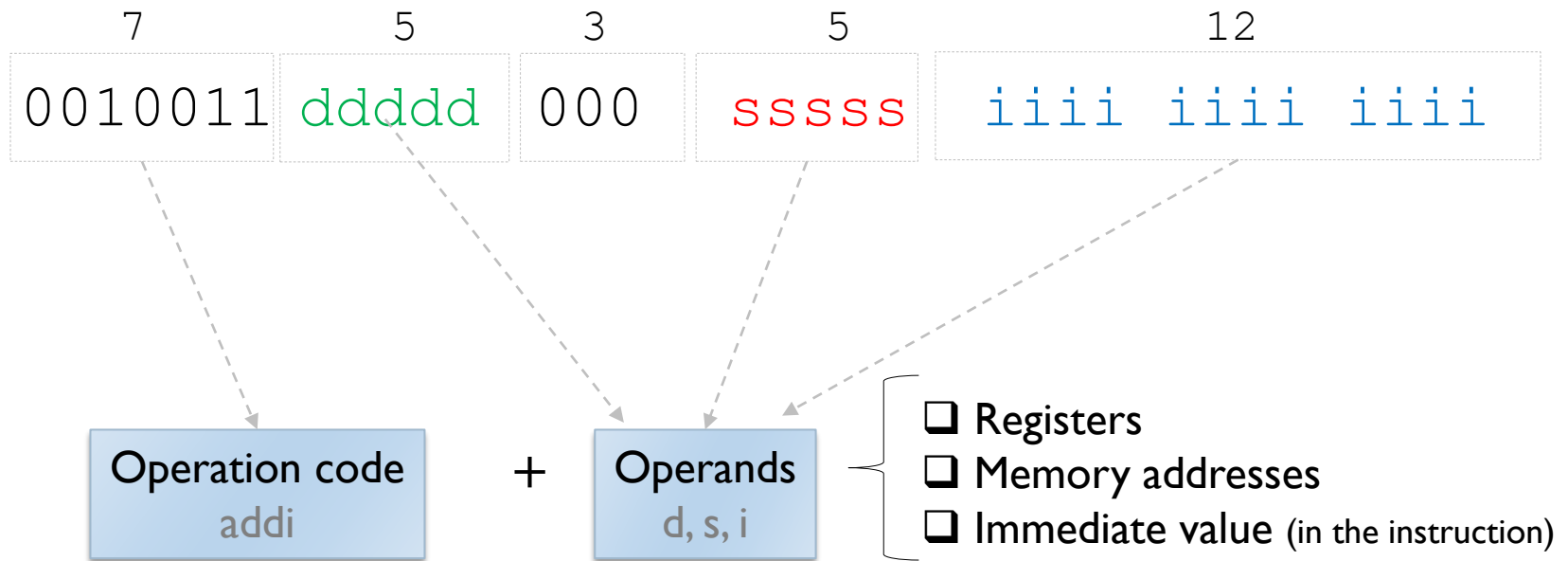Félix García-Carballeira, Alejandro Calderón Mateos
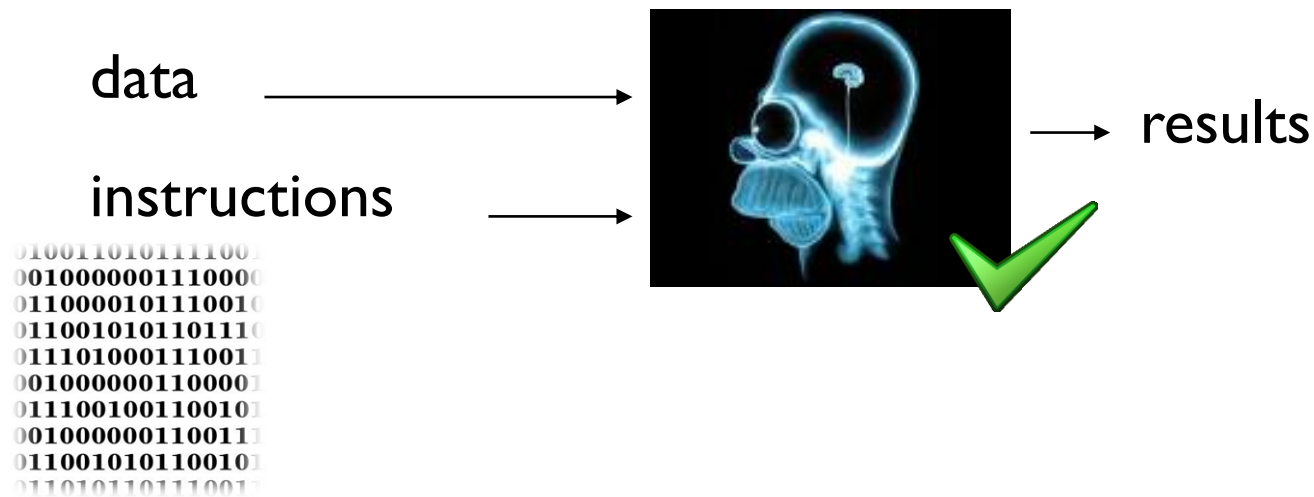
Machine instruction: definition

▸ **Machine instruction:** elementary operation that can be executed directly by the processor.

▸ Example: instruction of immediate add (addi) for 32 bits

  ▸ (d) = register (s) + immediate value (i)

7	5	3	5	12
0010011	ddddd	000	sssss	iiii iiii iiii

Operation code addi + **Operands** d, s, i
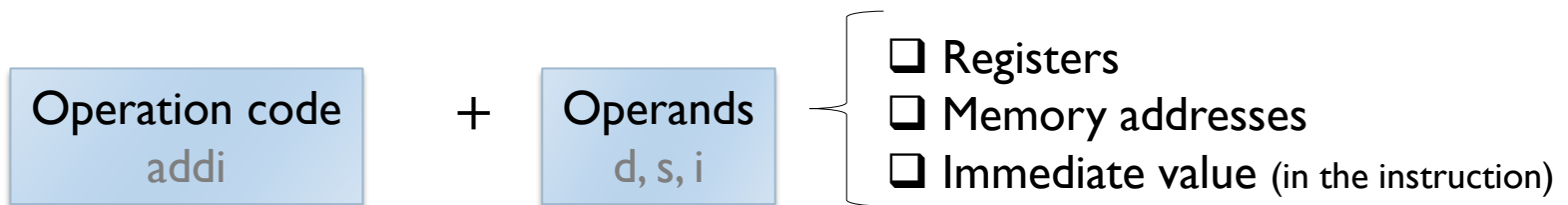
❑ Registers
❑ Memory addresses
❑ Immediate value (in the instruction)
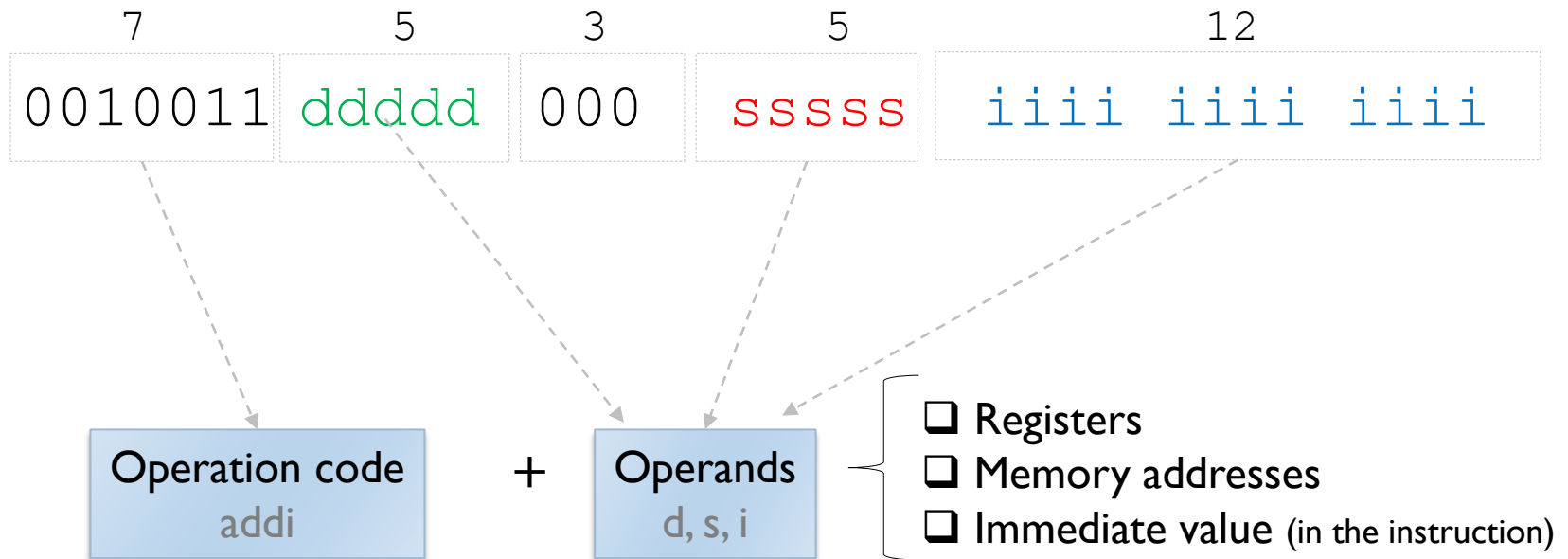
Machine instruction: properties

▸ Perform a single, simple task

▸ Operate on a fixed number of operands

▸ Include all the information necessary for its execution

7	5	3	5	12
0010011	ddddd	000	sssss	iiii iiii iiii

Operation code
addi

+

Operands
d, s, i

☐ Registers
☐ Memory addresses
☐ Immediate value (in the instruction)

Machine instruction: included information

- The operation to be performed.
- Where the operands are located:
  - In registers
  - In memory
  - In the instruction itself (immediate)
- Where to leave the results (as operand)
- A reference to the next instruction to be executed
  - Implicitly: the following instruction
    - A program is a consecutive sequence of machine instructions.
  - Explicitly in branching instructions (as operand)
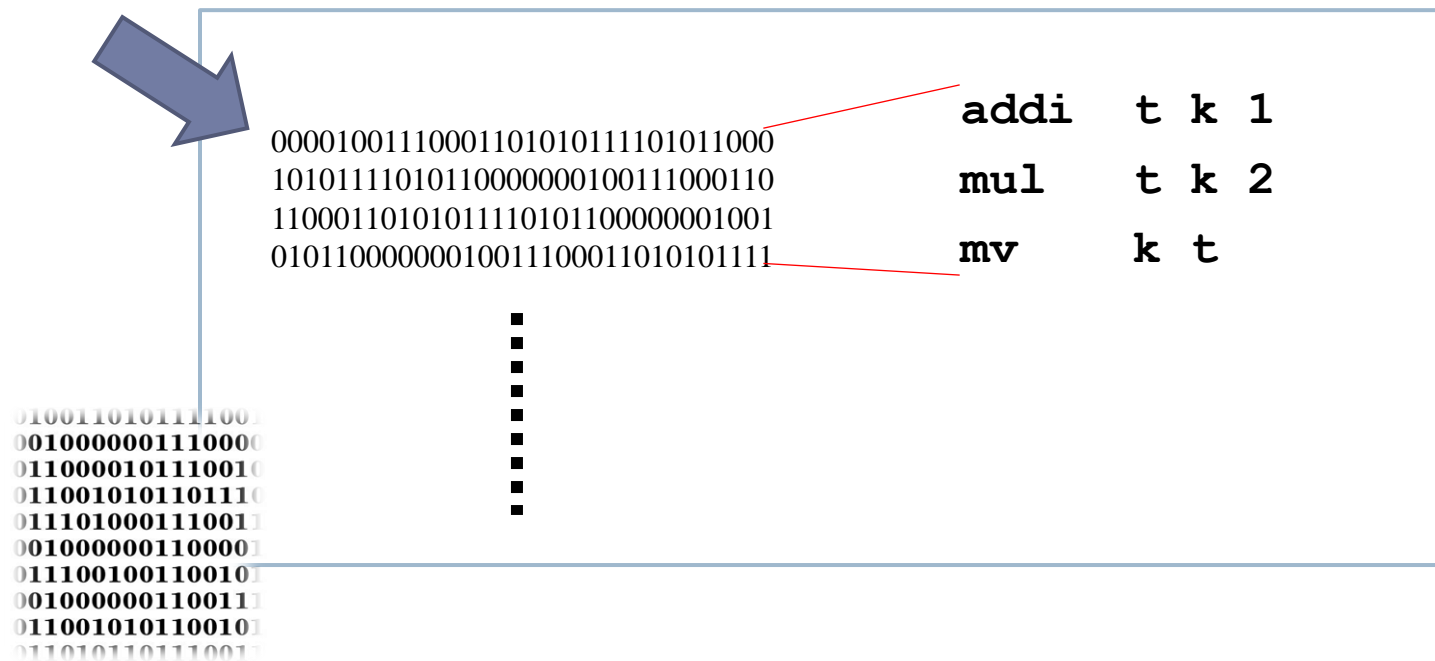
Operation code addi	+	Operands d, s, i	❑ Registers ❑ Memory addresses ❑ Immediate value (in the instruction)

Félix García-Carballeira, Alejandro Calderón Mateos

Types of information: instructions and data

‣ **What about the instructions?**
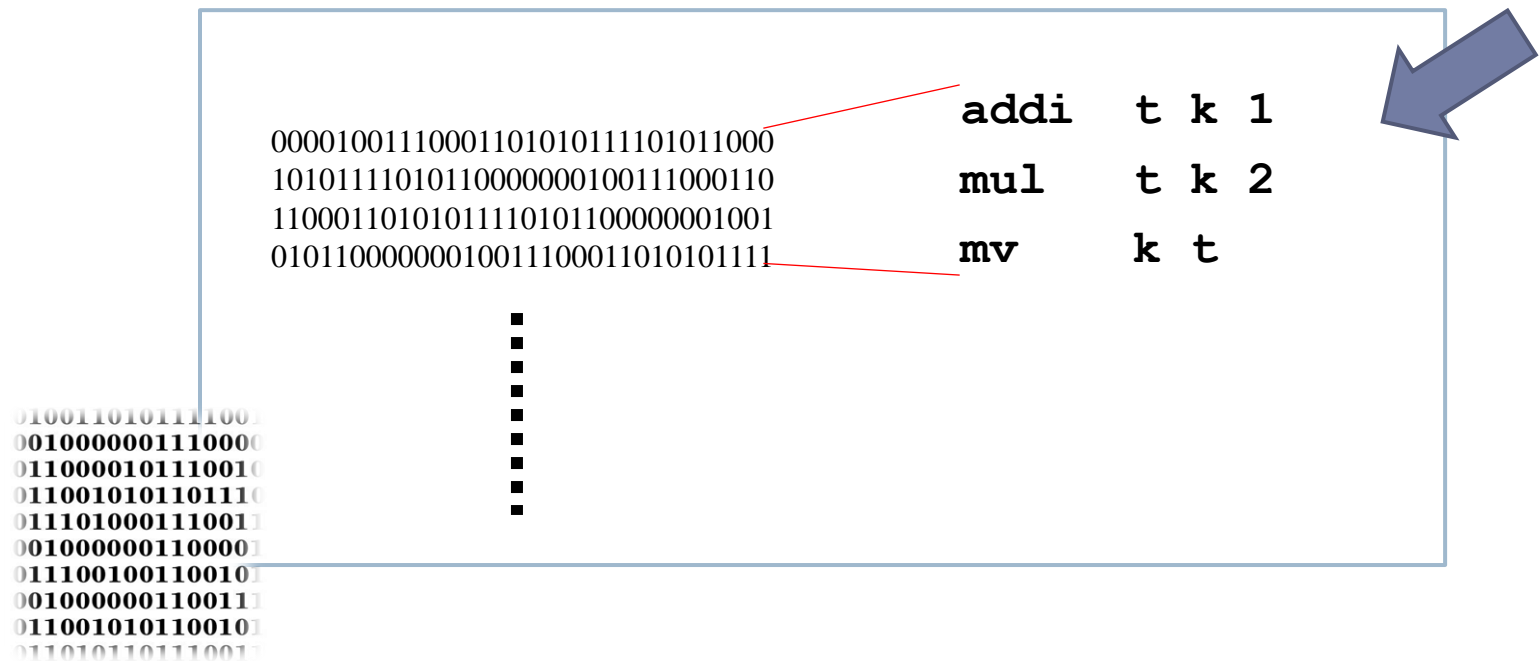
  ‣ Program, assembly language, ISA

data ⟶

instructions ⟶

Computer ⟶ results
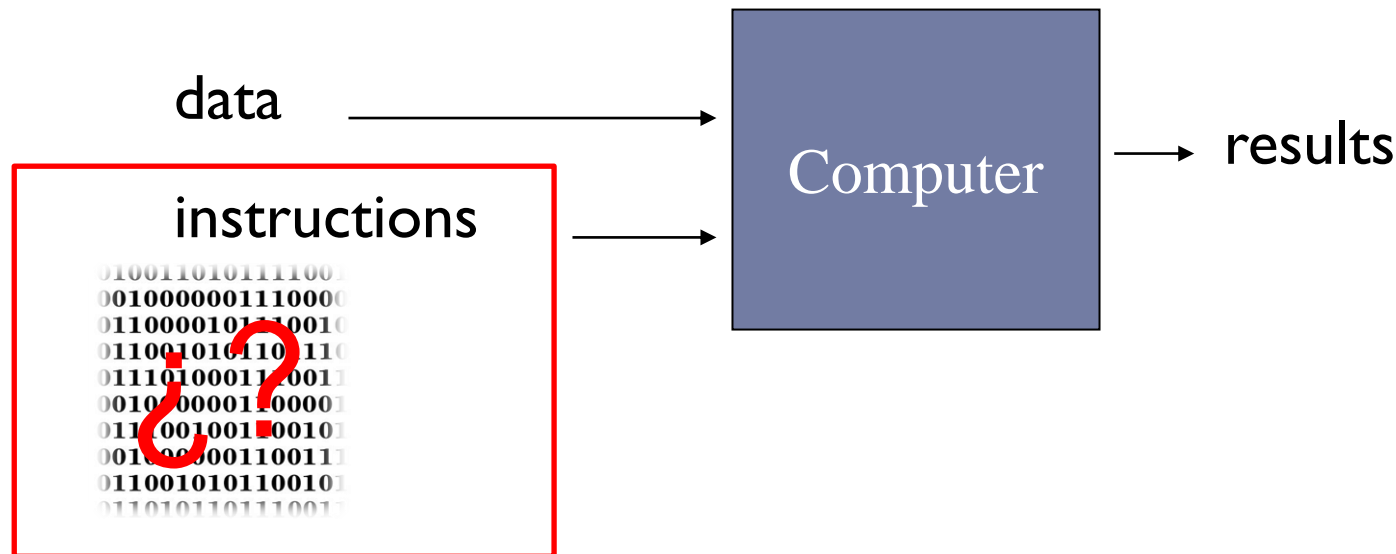
Definition of program

▸ **Program**: Ordered sequence of machine instructions that are executed by default in order.

```
000010011100011010101111101011000          addi    t k 1
101011110101100000010011100011 0           mul     t k 2
110001101010111101011000000010 01
010110000000100111000110101011 11          mv      k t
```

```
0100110101111003
0010000001110000
0110000101110010
0110010101101110
0111010001110011
0010000001100001
0111001001100101
0010000001100111
0110010101100101
0110101101110012
```

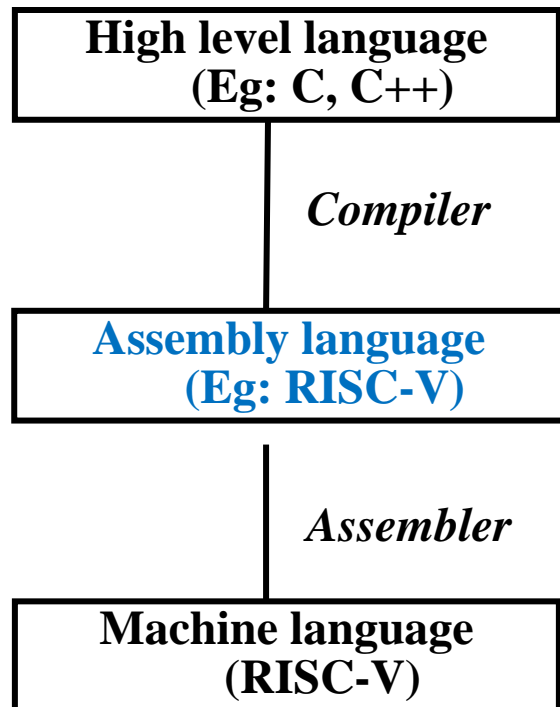Félix García-Carballeira,   Alejandro Calderón Mateos

Assembly language definition

▸ **Assembly language**: programmer-readable language that is the most direct representation of architecture-specific machine code.

```
00001001110001101010111101011000          addi    t k 1

10101111010110000000100111000110          mul     t k 2

11000110101011110101100000001001
                                          mv      k t
01011000000010011100011010101111
```

Félix García-Carballeira,   Alejandro Calderón Mateos

Assembly language definition

▸ **Assembly language**: programmer-readable language that is the most direct representation of architecture-specific machine code.

- ▸ Uses symbolic codes to represent instructions
  - ▸ `add` – addition
  - ▸ `lw` – Load a memory data

- ▸ Uses symbolic codes for data and references
  - ▸ `t0` – register

- ▸ There is an assembly instruction per machine instruction
  - ▸ `add t1, t2, t3`

Languages levels

```
High level language
(Eg: C, C++)
```

Compiler

```
Assembly language
(Eg: RISC-V)
```

Assembler

```
Machine language
(RISC-V)
```

temp = v[k];

v[k] = v[k+1];

v[k+1] = temp;

```
lw    t0, 0(x2)
lw    tl, 4(x2)
sw    tl, 0(x2)
sw    t0, 4(x2)
```

```
0000 1001 1100 0110 1010 1111 0101 1000
1010 1111 0101 1000 0000 1001 1100 0110
1100 0110 1010 1111 0101 1000 0000 1001
0101 1000 0000 1001 1100 0110 1010 1111
```

Instruction sets

- **Instruction Set Architecture** (ISA)
  - Instruction set of a processor
  - Boundary between hardware and software

- Examples:
  - 80x86
  - ARM
  - MIPS
  - RISC-V
  - PowerPC
  - Etc.

Characteristics of an instruction set (1/2)

▶ **Format and coding of the instruction set:**
  - ▶ Fixed or variable length instructions
    - ▸ 80x86: variable (from 1 up to 18 bytes)
    - ▸ MIPS, ARM: fixed

▶ **Operands:**
  - ▶ Registers, memory, the instruction itself

▶ **Type and size of operands:**
  - ▶ bytes: 8 bits
  - ▶ integers: 16, 32, 64 bits
  - ▶ floating-point numbers: single precision, double precision, etc.

▶ **Addressing modes:**
  - ▶ They specify where and how to access operands (register, memory or the instruction itself)

Characteristics of an instruction set (2/2)

‣ **Operations:**
  ‣ Arithmetic, logic, transfer, control, control, etc.

‣ **Flow control instructions:**
  ‣ Unconditional jumps
  ‣ Conditional jumps
  ‣ Procedure calls

‣ **Memory addressing:**
  ‣ Most of them use byte addressing
  ‣ They provide instructions for accessing multi-byte elements from a given position

Félix García-Carballeira,   Alejandro Calderón Mateos

Programming model of a computer

▸ A computer offers a programming model that consists of:

  ▸ Instruction set (assembly language)

    ▸ ISA: *Instruction Set Architecture*

    ▸ An instruction includes:

      ☐ Operation code

      ☐ Other elements: registers, memory address, numbers

  ▸ Storing elements

    ▸ Registers

    ▸ Memory

    ▸ Registers of I/O controllers

  ▸ Execution modes

Motivation to learn assembly

High-level language

Assembly language

Machine language

```
#include <stdio.h>

#define PI 3.1416
#define RADIO 20

int main ( )
{
  int l;

  l=2*PI*RADIO;
  printf("long: %d\n",l) ;
  return (0);
}
```

```
.data
    PI: .word 3.14156
    RADIO: .word 20

.text
    li   a0 2
    la   t0 PI
    lw   t0 ($t0)
    la   t1 RADIO
    lw   t1 (t1)
    mul a0 a0 t0
    mul a0 a0 t1

    li a7 1
    ecall
```

```
0100110101111001
0010000001110000
0110000101110010
0110010101101110
0111010001110011
0010000001100001
0111100100110010
0010000001100111
0110010101100101
0110101101110011
```
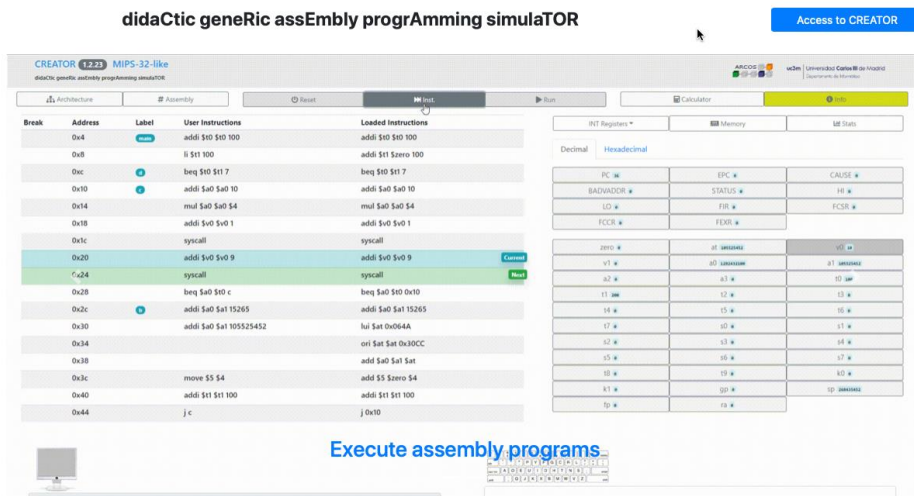
Félix García-Carballeira, Alejandro Calderón Mateos

Motivation to learn assembly

```
.data
    PI: .wo
    RADIUS

.text
    li   a0
    la   t0 P
    lw   t0 (t
    la   t1 R
    lw   t1 (t
    mul a0
    mul a0

    li a7 1
    ecall
```

```
#include <stdio.h>

#define PI 3.1416
#define RADIUS 20

int main ( )
{
    register int l;

    l=2*PI*RADIUS;
    printf("long: %d\n",l) ;
    return (0);
}
```
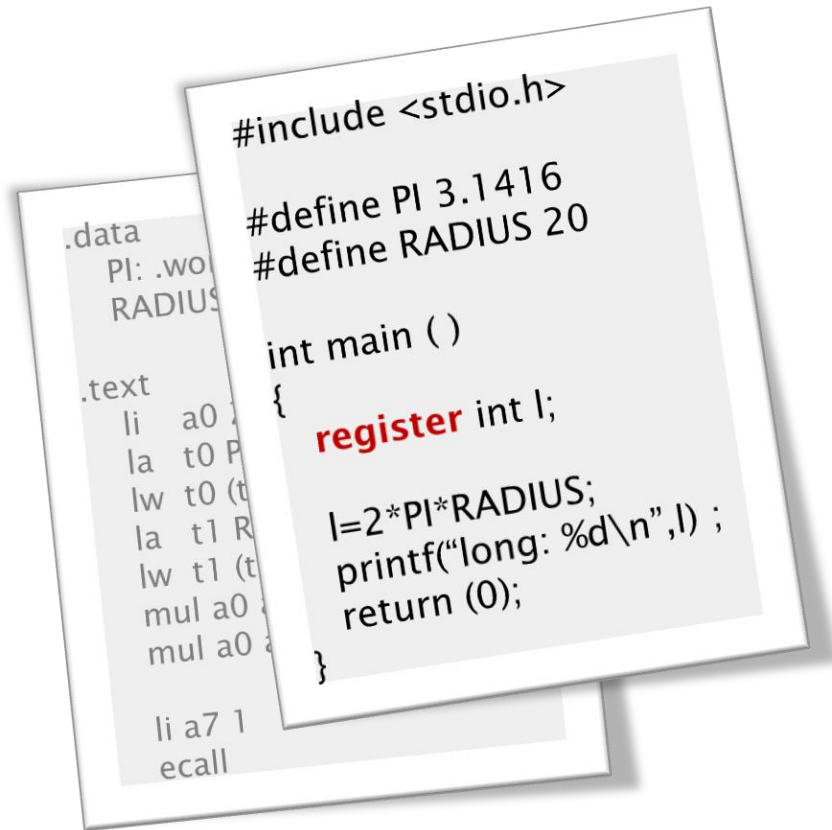
▸ Understand how high-level languages are executed
   ▸ C, C++, Python, Java, …

▸ Analyze the execution time of high-level instructions.

▸ Useful in specific domains:
   ▸ Compilers
   ▸ Operating Systems
   ▸ Games
   ▸ Embedded systems
   ▸ Etc.

Félix García-Carballeira,   Alejandro Calderón Mateos

Goals

▶ **Know how the elements of a high-level assembly language are represented.:**

   ▶ Data types (int, char, ...)
   ▶ Control structures (if, while, ...)

▶ **Be able to write small programs in assembler**

```
.data
    PI: .word 3.14156
    RADIUS: .word 20

.text
    li    a0 2
    la    t0 PI
    lw   t0 (t0)
    la    t1 RADIUS
    lw   t1 (t1)
    mul  a0 a0 t0
    mul  a0 a0 t1

    li a7 1
    ecall
```

Félix García-Carballeira,   Alejandro Calderón Mateos
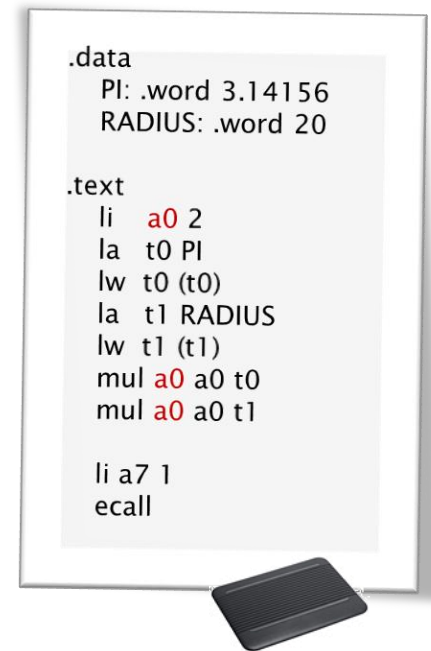
Motivation to use CREATOR simulator

didaCtic geneRic assEmbly progrAmming simulaTOR

https://creatorsim.github.io/

▸ **CREATOR: didaCtic geneRic assEmbly progrAmming simulaTOR**

▸ **CREATOR** can simulate MIPS$_{32}$ and RISC-V architectures

▸ **CREATOR** can be executed from Firefox, Chrome, Edge or Safari

Félix García-Carballeira,   Alejandro Calderón Mateos

Motivations to use RISC-V

➢ RISC processor (*Reduced Instruction Set Computer*)

➢ Examples of RISC processors:

- RISC-V, ARM, MIPS, etc.

https://pulp-platform.org/img/main/pulp_family.png
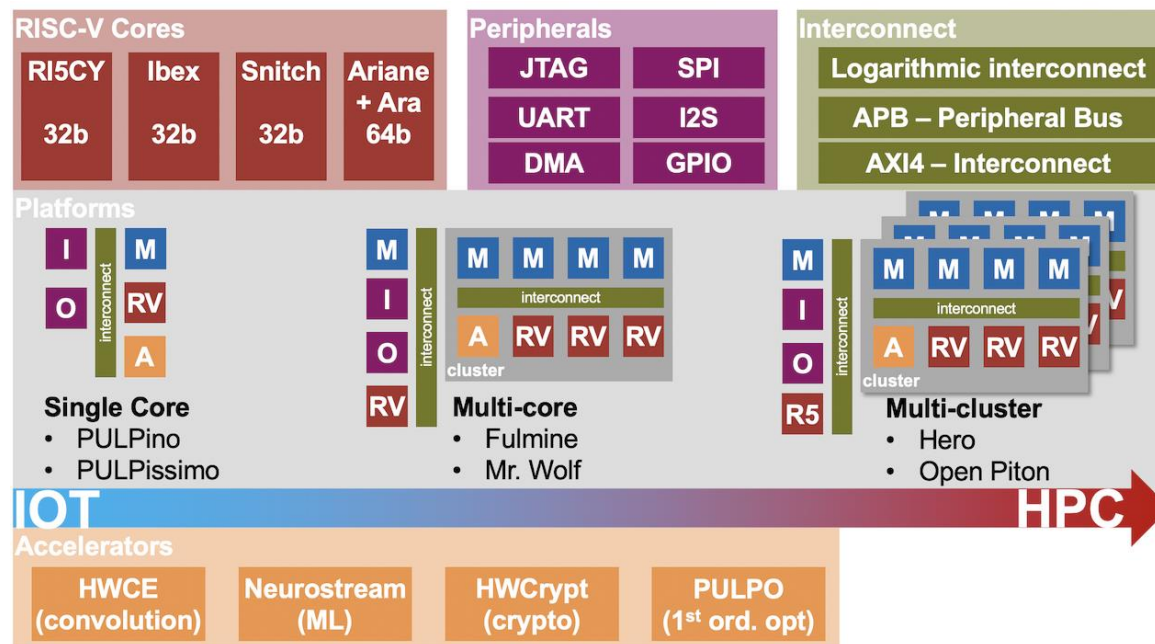
Advantages of using RISC-V

▶ Open hardware architecture:

  ▶ Allows anyone to design, manufacture and sell RISC-V chips and software

▶ Small and simple set of instructions

  ▶ RV32I -> ~47 instructions, RV32IMAF -> ~76

  ▶ Difference with x86 architecture instructions
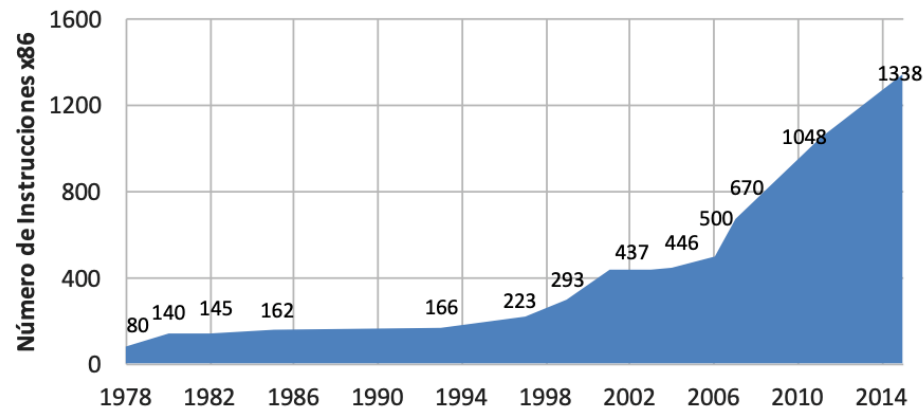
Guía Práctica de RISC-V.
David Patterson y Andrew Waterman

Félix García-Carballeira,   Alejandro Calderón Mateos

Contents

1. **Basic concepts on assembly programming**
   1. Motivations and goals
   2. Introduction to RISC-V32
2. RISC-V32 assembly language, memory model and data representation
3. Instruction formats and addressing modes
4. Procedure calls and stack convention

RISC-V instruction set

- Instruction sets:
  - RV32I: Set of instructions on integers. 32 bits
  - RV64I: Set of instructions on integers. 64 bits
  - RV128I: Set of instructions about integers. 128 bits
- On each of them there are different extensions:
  - M: instructions for integer multiplication and division
  - F: instructions for single-precision floating point
  - D: double precision floating-point instructions
  - G: Includes M, F and D
  - Q: quadruple-precision floating-point instructions
  - Etc.
- Example: RV64IF: RISC-V 64-bits processor with simple-precision floating-point instructions

RISC-V instruction sets to be described

- Instruction sets:
    - RV32I: Set of instructions on integers. 32 bits
    - RV64I: Set of instructions on integers. 64 bits
    - RV128I: Set of instructions on integers. 128 bits

- Each of them has different extensions:
    - M: instructions for integer multiplication and division
    - F: instructions for single-precision floating point
    - D: double precision floating-point instructions

RISC-V register file

https://www.uc3m.es/ss/Satellite/UC3MInstitucional/es/TextoMixta/1371208194039/Nueva_biblioteca_
de_Humanidades,_Comunicacion_y_Documentacion_%E2%80%9CCarmen_Martin_Gaite%E2%80%9D

ARCOS @ UC3M
Félix García-Carballeira,   Alejandro Calderón Mateos
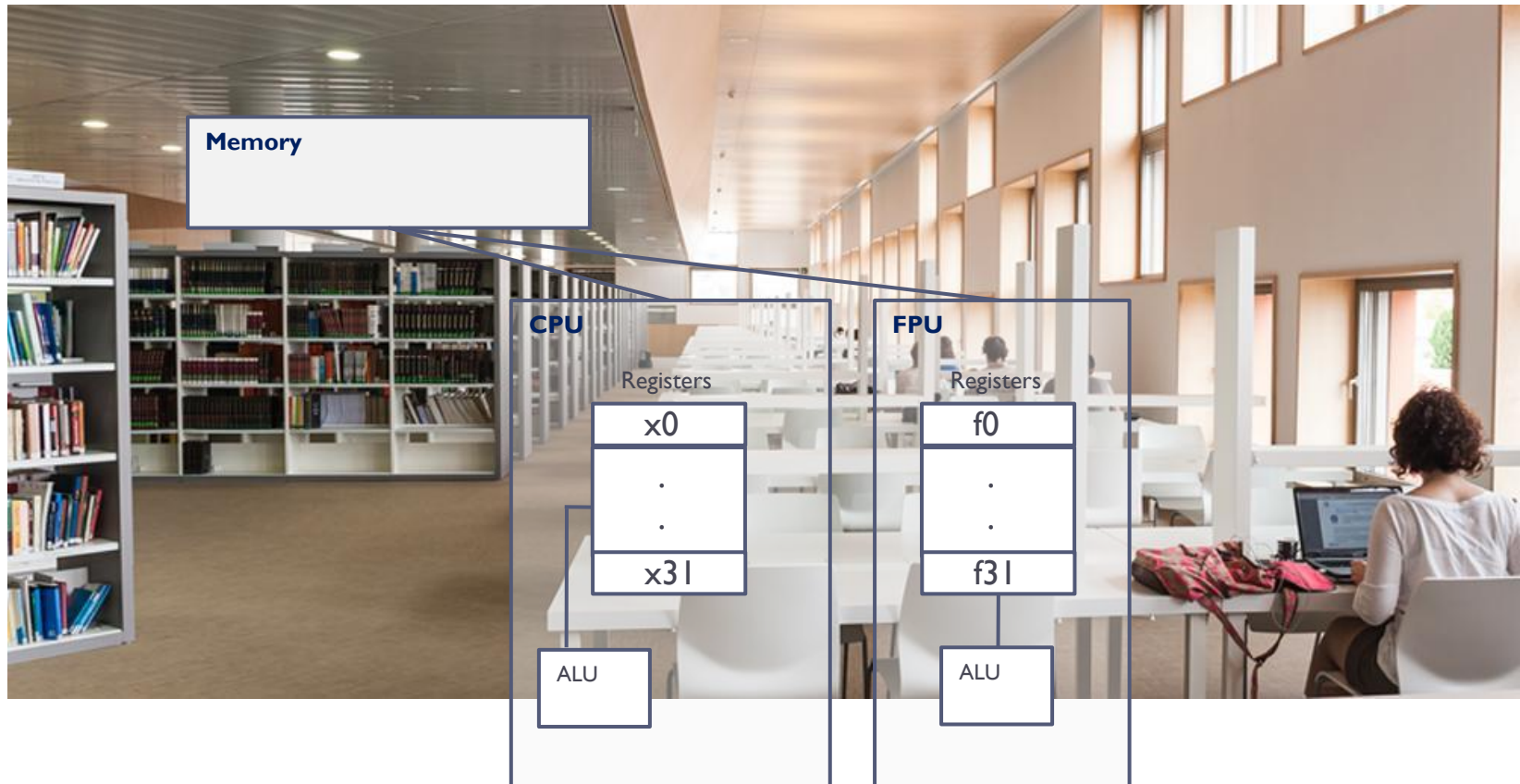
CREATOR

didaCtic geneRic assEmbly progrAmming simulaTOR

Register file

https://creatorsim.github.io/

ARCOS @ UC3M

Félix García-Carballeira,   Alejandro Calderón Mateos

Register File (integers)

ABI Name	Number	Uso
zero	x0	Constant 0
ra	x1	Return address (routines)
sp	x2	Stack pointer
gp	x3	Global pointer
tp	x4	Thread pointer
t0…t2	x5…x7	Temporary (NON-kept between calls)
s0/fp	x8	Temporary (kept between calls) / Frame stack pointer
s1	x9	Temporary (kept between calls)
a0…a1	x10…x11	Input argument for routines/returned values
a2…a7	x12…x17	Input argument for routines
s2…s11	x18…x27	Temporary (kept between calls)
t3…t6	x28…x31	Temporary (NON-kept between calls)

- There are 32 registers
  - Size of 4 bytes (one word)
  - Dual name:
    - Logical: ABI name (*Application Binary Interface*)
    - Numeric: starting with x at the beginning

- Use agreement
  - Reserved
  - Arguments
  - Results
  - Temporary
  - Pointers

Félix García-Carballeira,   Alejandro Calderón Mateos

Data transfer (integer registers)

▸ Copy data:

  ▸ Between registers

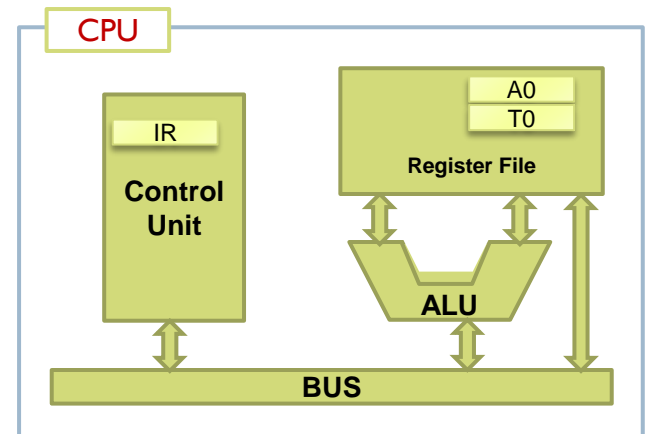 ▸ Between registers and memory (later)

▸ Examples:

  ▸ Copy from register to register
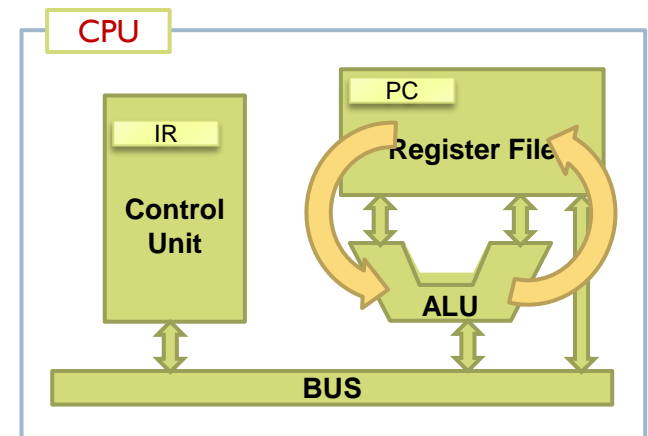   mv   a0 t0

  ▸ Immediate load
    li   t0   5

| mv | a0 t0 | # a0 ⟵ t0 |
| li | t0 5 | # t0 ⟵ 000....00101 |

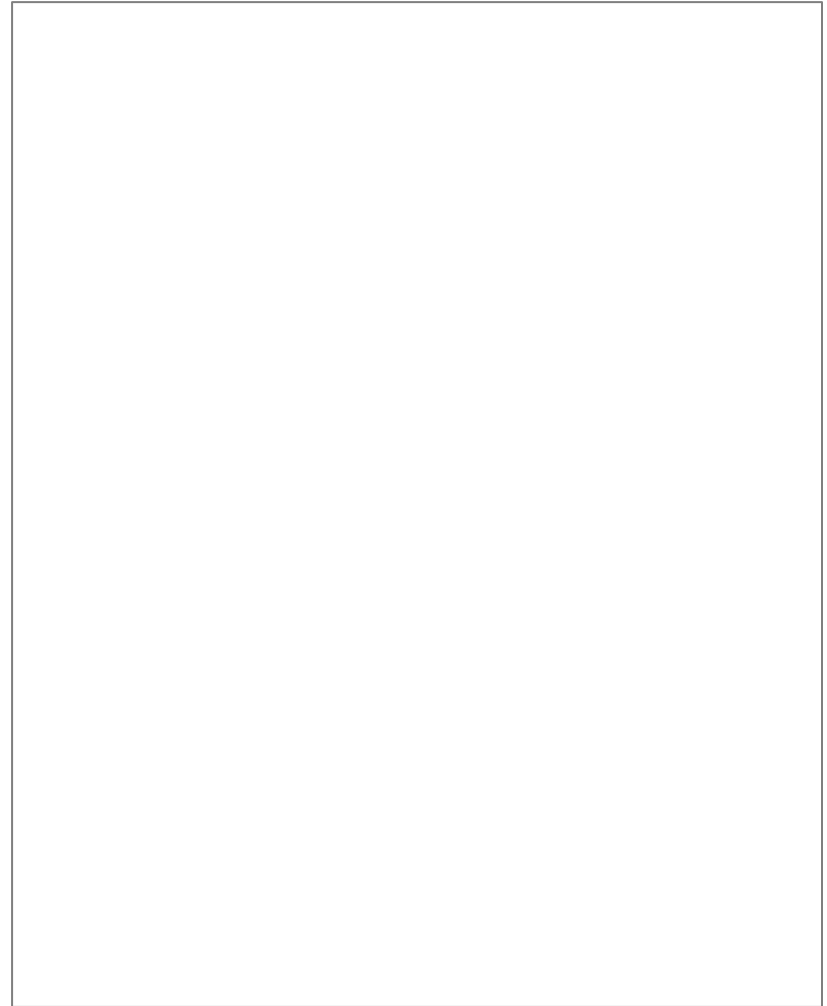# Arithmetic instructions (integer registers)

▸ Performs the arithmetic integer operations in two complement.

▸ Examples (ALU):

  ▸ Addition
    add   t0 t1 t2   # t0 ← t1 + t2
    addi t0 t1 5   # t0 ← t1 + 5

  ▸ Subtraction
    sub   t0 t1 t2   # t0 ← t1 - t2

  ▸ Multiplication
    mul   t0 t1 t2   # t0 ← t1 * t2

  ▸ Integer division (5 / 2=2)
    div   t0 t1 t2   # t0 ← t1 / t2

  ▸ Division remainder (5 % 2=1)
    rem   t0 t1 t2   # t0 ← t1 % t2

ARCOS @ UC3M

Félix García-Carballeira,   Alejandro Calderón Mateos

Example

```
int a = 5;
int b = 7;
int c = 8;
int i;
```

```
i = a * (b + c)
```

ARCOS @ UC3M

Félix García-Carballeira,   Alejandro Calderón Mateos

Example

```
int a = 5;
int b = 7;
int c = 8;
int i;
```

```
i = a * (b + c)
```

```
li  t1 5
li  t2 7
li  t3 8
```
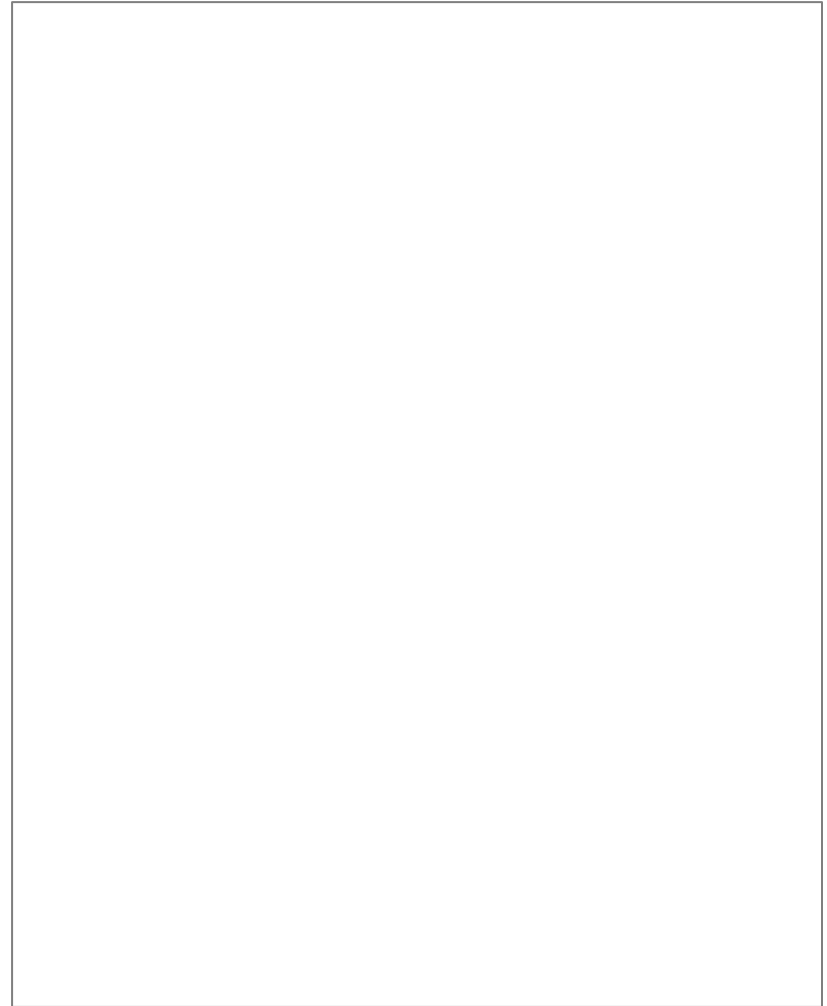
```
add   t4  t2  t3
mul   t4  t4  t1
```

ARCOS @ UC3M

Félix García-Carballeira,   Alejandro Calderón Mateos

Exercise

```
int a = 5;
int b = 7;
int c = 8;
int i;



i = -(a * (b - 10) + c)
```

ARCOS @ UC3M

Félix García-Carballeira, Alejandro Calderón Mateos

Exercise (solution)

```
int a = 5;

int b = 7;

int c = 8;

int i;
```
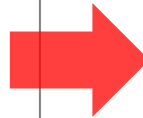
```
i = -(a * (b – 10) + c)
```

```
li   t1 5
li   t2 7
li   t3 8
```

```
li    t0 10
sub   t4  t2  t0
mul   t4  t4  t1
add   t4  t4  t3
li    t0 -1
mul   t4  t4  t0
```

ARCOS @ UC3M

Félix García-Carballeira,   Alejandro Calderón Mateos

Exercise

```
li t1 5
li t2 7
li t3 8

li   t0 10
sub  t4 t2 t0
mul  t4 t4 t1
add  t4 t4 t3
li   t0 -1
mul  t4 t4 t0
```
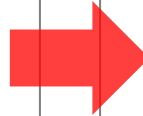
Can be performed by using less instrucciones?

ARCOS @ UC3M

Félix García-Carballeira,   Alejandro Calderón Mateos

Exercise (solution)

```
li t1 5
li t2 7
li t3 8

li    t0 10
sub   t4 t2 t0
mul   t4 t4 t1
add   t4 t4 t3
li    t0 -1
mul   t4 t4 t0
```

```
li t1 5
li t2 7
li t3 8

addi  t4 t2 -10
mul   t4 t4 t1
add   t4 t4 t3
add   t4 x0 t4
```

ARCOS @ UC3M

Félix García-Carballeira,   Alejandro Calderón Mateos

Register File (floating-point)

ABI Name	Name	Usage
ft0…ft7	f0 … f7	Temporals (like t…)
fs0…fs1	f8 … f9	There are saved (like s…)
fa0…fa1	f10 … f11	Arguments/return (like a…)
fa2…fa7	f12 … f17	Arguments (like a…)
fs2…fs11	f18 … f27	There are saved (like s…)
ft8…ft11	f28 … f31	Temporals (like t…)

▸ There are 32 registers

▸ The ft0 register does not have its value set to 0.

▸ In the single precision extension, the registers are 32 bits (4 bytes).

▸ In the double precision extension, the registers are 64 bits (8 bytes) and can store:

  ▸ Single precision values in the lower 32 bits of the register

  ▸ Double precision values in the 64 bits of the register

Floating-point register file

▸ **There are 32 floating-point registers in addition to the integer registers:**
  ▸ From f0 to f31

▸ **Copy registers (.s .d):**
  ▸ fmv.s   rd rs   # rd = rs
▸ **Common arithmetic operations (.s .d):**
  ▸ fadd.s   rd rs1 rs2   # rd = rs1 + rs2
  ▸ fsub.s   rd rs1 rs2   # rd = rs1 - rs2
  ▸ fmul.s   rd rs1 rs2   # rd = rs1 * rs2
  ▸ fdiv.s   rd rs1 rs2   # rd = rs1 / rs2
  ▸ fmin.s   rd rs1 rs2   # rd = min(rs1, rs2)
  ▸ fmax.s rd rs1 rs2   # rd = max(rs1, rs2)
  ▸ fsqrt.s rd rs1   # rd = sqrt(rs1)
  ▸ fmadd.s   rd rs1 rs2 rs3 # rd = rs1 x rs2 + rs3
  ▸ fmsub.s   rd rs1 rs2 rs3 # rd = rs1 x rs2 - rs3
  ▸ fabs.s   rd rs   # rd = |rs|
  ▸ fneg.s   rd rs   # rd = -rs

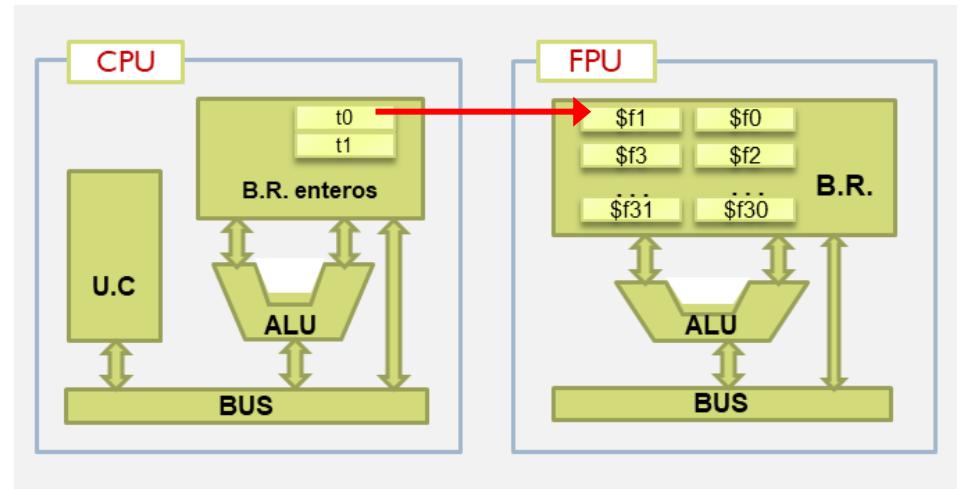FPU

| f1 | f0 |
| f3 | f2 |
| . . . | . . . | R.F.
| f31 | f30 |

ALU

BUS
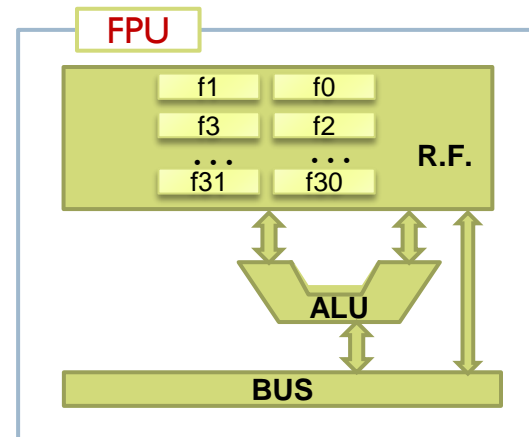
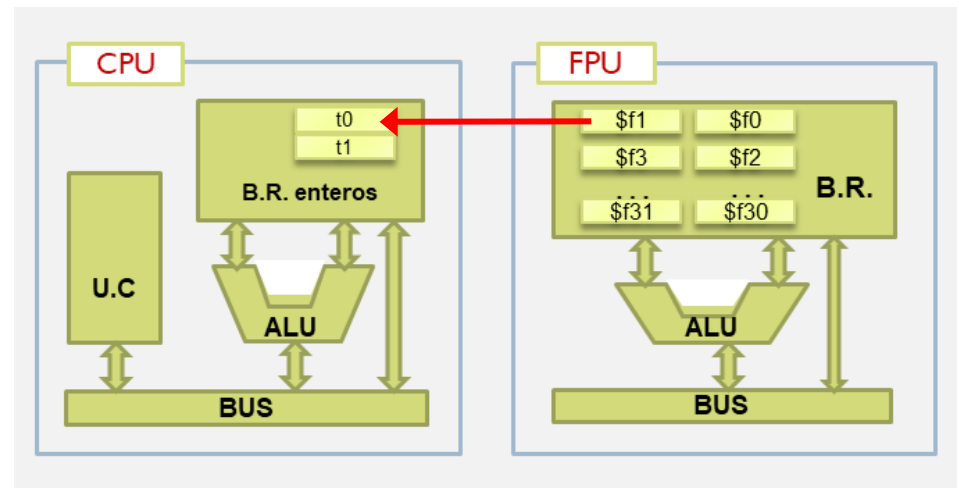# Copy instructions
(integer registers<-> floating-point registers)

fmv.w.x rd rs

▸ Copy from integer register rs to a floating register rd (*single precisión*)

fmv.x.w rd rs

▸ Copy from floating register rs (*single precision*) to a integer register rd

Conversion operations (1/3)

integer <-> simple precision

- `fcvt.w.s  rd, rs`
  - Convert from simple precision (value in floating register rs) to a 32-bit integer 32 **with** sign (integer register rd).

- `fcvt.wu.s rd, rs`
  - Convert from simple precision (value in floating register rs) to a 32-bit integer 32 **withou** sign (integer register rd).

- `fcvt.s.w  rd, rs`
  - Convert a 32-bit integer 32 **with** sign (value in integer register rs) to a simple precision (floating register rd).

- `fcvt.s.wu rd, rs`
  - Convert a 32-bit integer 32 **without** sign (value in integer register rs) to a simple precision (floating register rd).

Félix García-Carballeira,   Alejandro Calderón Mateos

# Conversion operations (2/3)
integer <-> double precision

- `fcvt.w.d  rd, rs`
  - Convert from double precision (value in floating register rs) to a 32-bit integer 32 **with** sign (integer register rd).

- `fcvt.wu.d rd, rs`
  - Convert from double precision (value in floating register rs) to a 32-bit integer 32 **withou** sign (integer register rd).

- `fcvt.d.w  rd, rs`
  - Convert a 32-bit integer 32 **with** sign (value in integer register rs) to a double precision (floating register rd).

- `fcvt.d.wu rd, rs1`
  - Convert a 32-bit integer 32 **without** sign (value in integer register rs) to a double precision (floating register rd).

Félix García-Carballeira,   Alejandro Calderón Mateos

Conversion operations (3/3)

double precision <-> simple precision

- ### fcvt.s.d rd, rs1

  - Convert from double precision (value in floating register rs) to simple precision (floating register rd).

- ### fcvt.d.s rd, rs

  - Convert from simple precision (value in floating register rs) to double precision (floating register rd).

Classification of floating-point numbers

▸ `fclass.s rd, rs1`   (simple precision)

▸ `fclass.d rd, rs1`   (doble precision)

▸ Save in `rd` the type of the floating-point number of the register `rs1`:

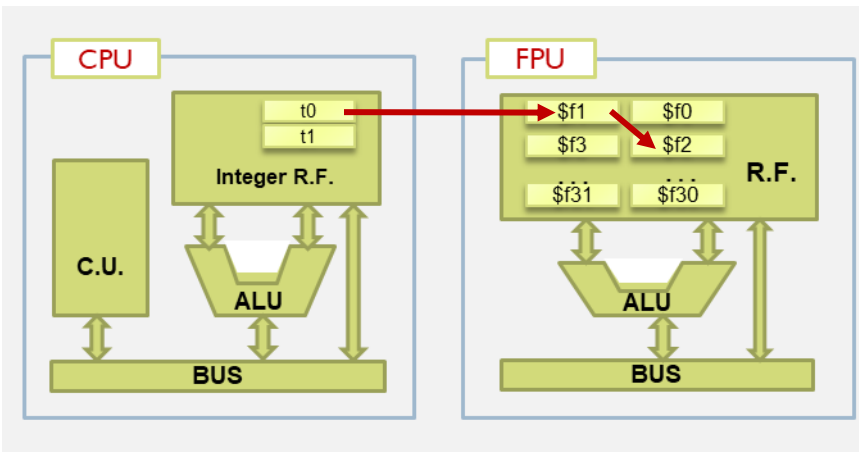| Value in rd	Meaning
0	-Inf
1	Negative normalized number
2	Negative non-normalized number
3	-0
4	+0
5	Positive non-normalized number
6	Positive normalized number
7	+Inf
8	NaN (non-quiet)
9	NaN (quit)

Example

```
float PI    = 3,1415;

int   radio = 4;

float length;


length = PI * radio;
```



```
.text
main:

  li t0, 0x40490E56
     # does not exist li.s
     # 0x40490E56 is the
     # representation IEEE754
     # in hexadecimal for 3.1415
  fmv.w.x    ft0, t0   # ft0 ← t0
  li         t1    4   # 4 in Ca2
  fcvt.s.w   ft1, t1   # 4 in ieee754
  fmul.s     ft0, ft0, ft1
```

ARCOS @ UC3M

Félix García-Carballeira,   Alejandro Calderón Mateos

Logical instructions

▸ **Boolean** operations

▸ Examples:

    ▸ AND
    and t0 t1 t2   (t0 = t1 & t2)
    andi t0 t1 t2   (t0 = t1 & t2)

    ▸ OR
    or t0 t1 t2   (t0 = t1 | t2)
    ori t0 t1 80   (t0 = t1 | 80)

    ▸ NOT
    not t0 t1   (t0 = ! t1)
        xori t0 t1 -1

    ▸ XOR
    xor t0 t1 t2   (t0 = t1 ^ t2)

AND	1100
	1010
	1000

OR	1100
	1010
	1110

NOT	10
	01

XOR	1100
	1010
	0110

Exercise

```
li t0, 5
li t1, 8

and t2, t1, t0
```
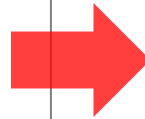
What will be the value of t2?

Félix García-Carballeira,   Alejandro Calderón Mateos

Exercise (solution)

```
    li t0, 5
    li t1, 8

    and t2, t1, t0
```
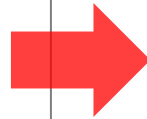
```
      00...0101   t0
      00...1000   t1
      ─────────────
and   00...0000   t2
```

ARCOS @ UC3M

Félix García-Carballeira, Alejandro Calderón Mateos

Exercise

```
li t0, 5
li t1, 0x007FFFFF


and t2, t1, t0
```
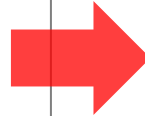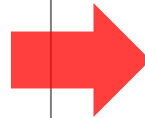
What does an "and" with 0x007FFFFF allow to do?

Exercise (solution)

```
li t0, 5
li t1, 0x007FFFFF

and t2, t1, t0
```

What does an "and" with 0x007FFFFF allow to do?

Obtain the 23 least significant bits

The constant used for bit selection is called a mask.
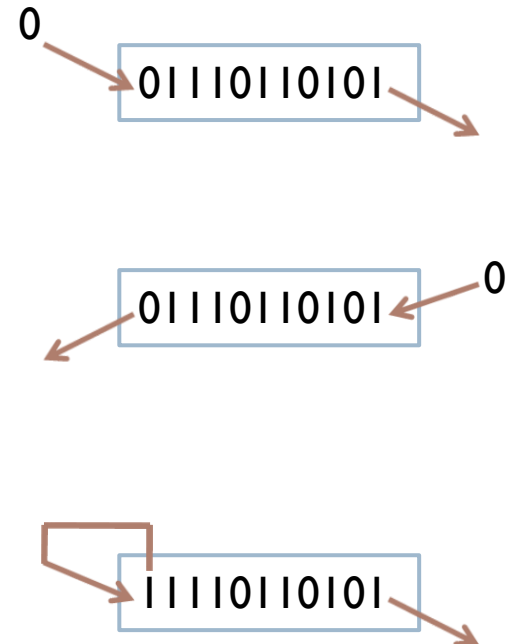
Shift instructions

▸ **Bits movement**

▸ **Examples:**

   ▸ Shift right logical
srl   t0 t0 4   (t0 = t0 >> 4 bits)

   ▸ Shift left logical
sll   t0 t0 5   (t0 = t0 << 5 bits)

   ▸ Shift right arithmetic
sra t0 t0 2   (t0 = t0 >> 2 bits)

0
0110110101

0110110101 ← 0

1110110101

Félix García-Carballeira, Alejandro Calderón Mateos
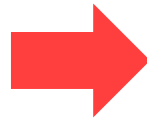
Exercise (solution)

```
li t0, 5
li t1, 6

srai t0, t1, 1

slli t0, t1, 1
```

- What is the value of t0?

  000 …. 0110 t1
  shift one bit to right (/2)
  000 ….. 0011   t0

- What is the value of $t0?

  000 …. 0110 t1
  Shit one bit to left (x2)
  000 ….. 1100   t0

Félix García-Carballeira,   Alejandro Calderón Mateos

Exercise

Make a program that detects the sign of a stored number t0 and leaves in t1 a 1 if it is negative and a 0 if it is positive.

ARCOS @ UC3M

Félix García-Carballeira, Alejandro Calderón Mateos

Exercise (solution)

Make a program that detects the sign of a stored number t0 and leaves in t1 a 1 if it is negative and a 0 if it is positive.

```
 li   t0 -3

srli t1 t0 31
```

Comparison instructions (integer registers)

▸ slt   rd, rs1, rs2   if (s(rs1) < s(rs2))   rd = 1; else rd = 0

▸ sltu rd, rs1, rs2   if (u(rs1) < u(rs2))   rd = 1; else rd = 0

▸ slti   rd, rs1,   5   if (s(rs1) <   s(5))   rd = 1; else rd = 0

▸ sltiu rd, rs1,   5   if (u(rs1) < u(5))   rd = 1; else rd = 0

▸ seqz rd, rs1   if (rs1 == 0)   rd = 1; else rd = 0

▸ snez rd, rs1   if (rs1 != 0)   rd = 1; else rd = 0

▸ sgtz rd, rs1   if (rs1 > 0)   rd = 1; else rd = 0

▸ sltz rd, rs1   if (rs1 < 0)   rd = 1; else rd = 0

# Comparison instructions (floating-point registers)

▸ **Simple precision**

- ▸ feq.s   rd, rs1, rs2   if (rs1== rs2)   rd= 1; else rd = 0
- ▸ fle.s   rd, rs1, rs2   if (rs1<= rs2)   rd= 1; else rd = 0
- ▸ flt.s   rd, rs1, rs2   if (rs1<   rs2)   rd= 1; else rd = 0

▸ **Doble precision:**

- ▸ feq.d   rd, rs1, rs2   if (rs1== rs2)   rd= 1; else rd = 0
- ▸ fle.d   rd, rs1, rs2   if (rs1<= rs2)   rd= 1; else rd = 0
- ▸ flt.d   rd, rs1, rs2   if (rs1<   rs2)   rd= 1; else rd = 0

Control flow

▶ Change the sequence of instructions to be executed

▶ Several types:

   ▶ Conditional branches:
      ▶ Branch if value match condition
      ▶ E.g.: bne t0 t1 label1

   ▶ Unconditional branches:
      ▶ Always branch
        E.g.: j etiqueta2

   ▶ Function calls:
      ▶ Branch with return
      ▶ E.g.: jal ra subrutina1   ……   jr ra
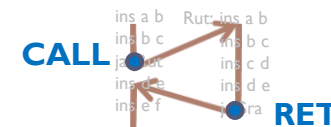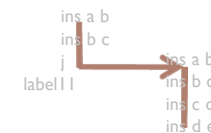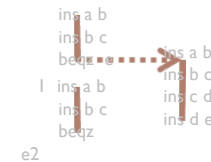
Branch instructions

- Conditional (only with integer registers):
  - `beq    t0   t1   label1    # jump to label1 if t0 == t1`
  - `bne    t0   t1   label1    # jump to label1 if t0 != t1`
  - `blt    t0   t1   label1    # jump to label1 if t0 <  t1`
  - `bltu   t0   t1   label1    # jump to label1 if t0 <  t1 (unsigned)`
  - `bge    t0   t1   label1    # jump to label1 if t0 >= t1`
  - `bgeu   t0   t1   label1    # jump to label1 if t0 >= t1 (unsigned)`
  - `(as pseudo-instructions)`
  - `bgt    t0   t1   label1    # jump to label1 if t0 >  t1`
  - `ble    t0   t1   label1    # jump to label1 if t0 <= t1`

Branch instructions
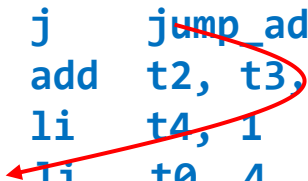jump label

▸ Conditional (only with integer registers):
- ▸ `beq    t0   t1   label1   # jump to label1 if t0 == t1`
- ▸ `bne    t0   t1   label1   # jump to label1 if t0 != t1`
- ▸ `blt    t0   t1   label1   # jump to label1 if t0 <  t1`
- ▸ `bltu   t0   t1   label1   # jump to label1 if t0 <  t1 (unsigned)`
- ▸ `bge    t0   t1   label1   # jump to label1 if t0 >= t1`
- ▸ `bgeu   t0   t1   label1   # jump to label1 if t0 >= t1 (unsigned)`

- ▸ `bgt    t0   t1   label1   # jump to label1 if t0 >  t1`
- ▸ `ble    t0   t1   label1   # jump to label1 if t0 <= t1`

▸ Unconditional:
- ▸ `j      label1   # jump to label1. Equivalent to beq x0 x0 label1`

    label1 refers to an instruction (represents a memory address
    where the instruction is located) which is skipped:

```
                 add   t1, t2, t3
                 j     jump_addr
                 add   t2, t3, t4
                 li    t4, 1
jump_addr:       li    t0, 4
```

Control flow
if

```
beq    t1  = t0
bne    t1 != t0
bge    t1 >= t0
ble    t0 <= t1
blt    t1 <  t0
bgt    t0 >  t1
```

```c
int a=1;

int b=2;


main ()

{

  if (a < b) {

    a = b;

  }

  ...

}
```

ARCOS @ UC3M

Félix García-Carballeira, Alejandro Calderón Mateos

Control flow
if

CREATOR
WᴇᴅSIM

beq	t1 = t0
bne	t1 != t0
bge	t1 >= t0
ble	t0 <= t1
blt	t1 < t0
bgt	t0 > t1

```
int a=1;

int b=2;


main ()

{

  if (a < b) {

    a = b;

  }

  ...

}
```

```
          li   t1 1
          li   t2 2

if_1:    blt t1 t2 then_1
          j    fin_1

then_1: mv t1 t2

fin_1:  ...
```

ARCOS @ UC3M

Félix García-Carballeira, Alejandro Calderón Mateos

Control flow
if

beq   t1 = t0
bne   t1 != t0
bge   t1 >= t0
ble   t0 <= t1
blt   t1 <   t0
bgt   t0 >   t1

CREATOR

WEPSIM

```
int a=1;

int b=2;


main ()

{

  if (a < b) {

    a = b;

  }

  ...

}
```

```
              li   t1 1
              li   t2 2


if_2:    bge t1 t2 fin_2


then_2: mv t1 t2



fin_2: ...
```

ARCOS @ UC3M

Félix García-Carballeira,   Alejandro Calderón Mateos

Control flow
if-else

CREATOR

WebSIM

beq	t1 = t0
bne	t1 != t0
bge	t1 >= t0
ble	t0 <= t1
blt	t1 < t0
bgt	t0 > t1

```
int a=1;

int b=2;


main ()

{

  if (a < b){

      // action 1

  } else {

      // action 2

  }

}
```

```
              li   t1 1
              li   t2 2


if_3:     bge t1 t2 else_3


then_3: # action 1

          j    fi_3


else_3: # action 2


fi_3: ...
```

ARCOS @ UC3M

Félix García-Carballeira,   Alejandro Calderón Mateos

Exercise

```
int b1 = 4;
int b2 = 2;


if (b2 == 8) {
    b1 = 1;
}
...
```

ARCOS @ UC3M

Félix García-Carballeira, Alejandro Calderón Mateos
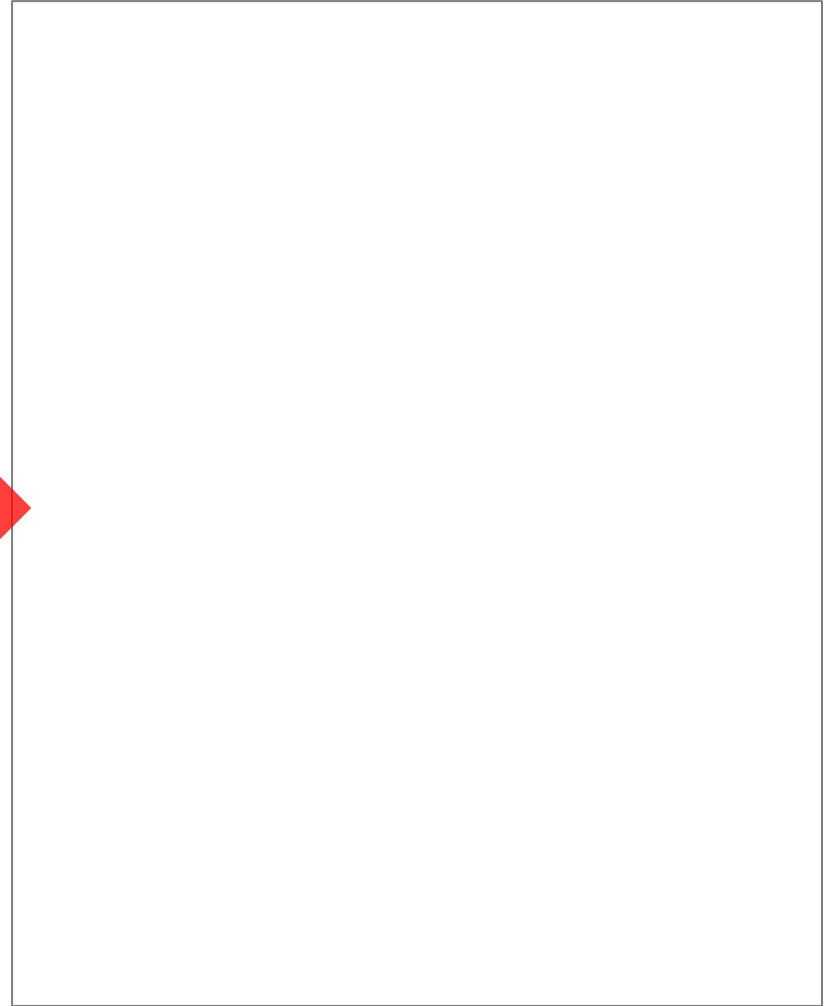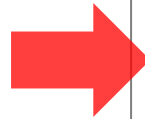
Exercise (solution)

```
int b1 = 4;
int b2 = 2;


 if (b2 == 8) {
     b1 = 1;
 }
 ...
```
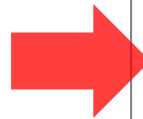
```
        li    t0 4
        li    t1 2
        li    t2 8

        bne   t0 t2   fin1
        li    t1 1
fin1:   ...
```

ARCOS @ UC3M

Félix García-Carballeira,   Alejandro Calderón Mateos

Branches with floating-point numbers

Jump to label if

ft1 < ft2

```
        flt t0, ft1, ft2
        bne t0, x0, label
        . . .
label:
```

ARCOS @ UC3M

Félix García-Carballeira,   Alejandro Calderón Mateos

Control flow while

```
beq    t1  = t0
bne    t1 != t0
bge    t1 >= t0
ble    t0 <= t1
blt    t1 <  t0
bgt    t0 >  t1
```

```c
int i;


main ()
{
   i=0;
   while (i < 10) {

     /* action */

     i = i + 1 ;

   }
}
```

ARCOS @ UC3M

Félix García-Carballeira,   Alejandro Calderón Mateos

Control flow while

CREATOR

WepSIM

```
beq    t1  = t0
bne    t1 != t0
bge    t1 >= t0
ble    t0 <= t1
blt    t1 <  t0
bgt    t0 >  t1
```

```c
int i;


main ()
{
    i=0;
    while (i < 10) {

      /* action */

       i = i + 1 ;
    }
}
```
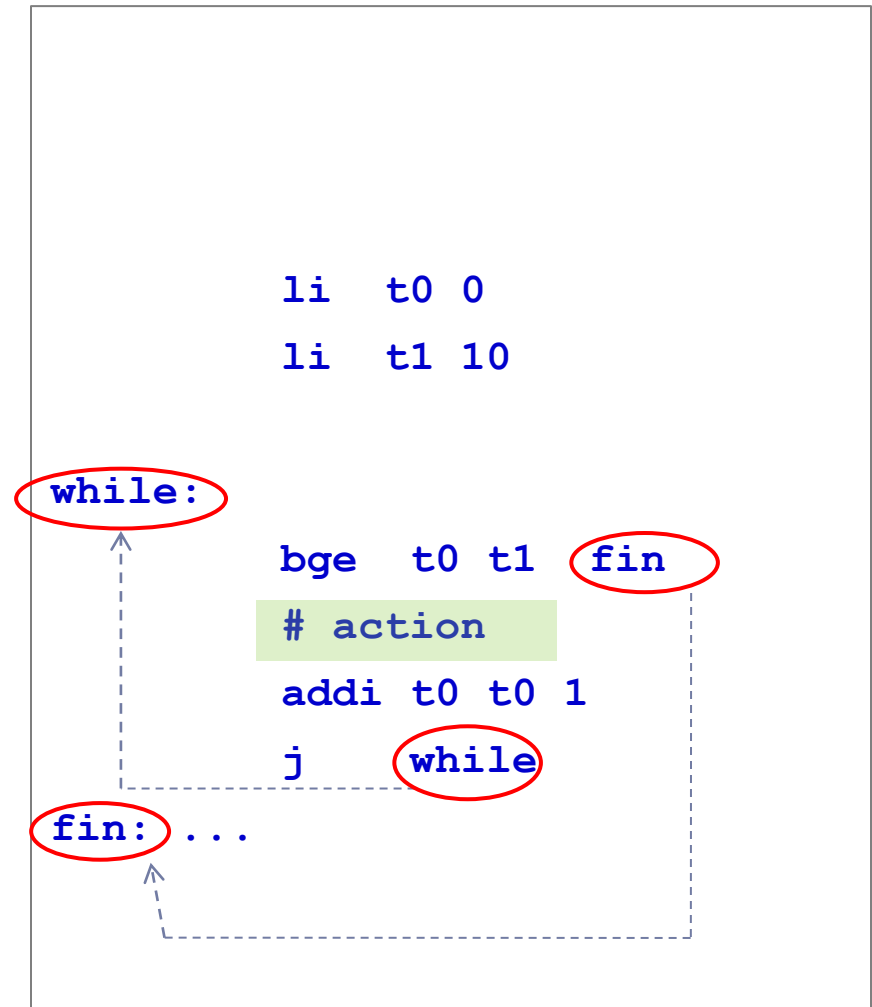
```
            li   t0 0
            li   t1 10

while:

            bge  t0 t1  fin
            # action
            addi t0 t0 1
            j    while
fin: ...
```

Félix García-Carballeira, Alejandro Calderón Mateos

Exercise

Make a program that calculates the sum of the first ten numbers and leaves this value in register a0

ARCOS @ UC3M

Félix García-Carballeira,   Alejandro Calderón Mateos
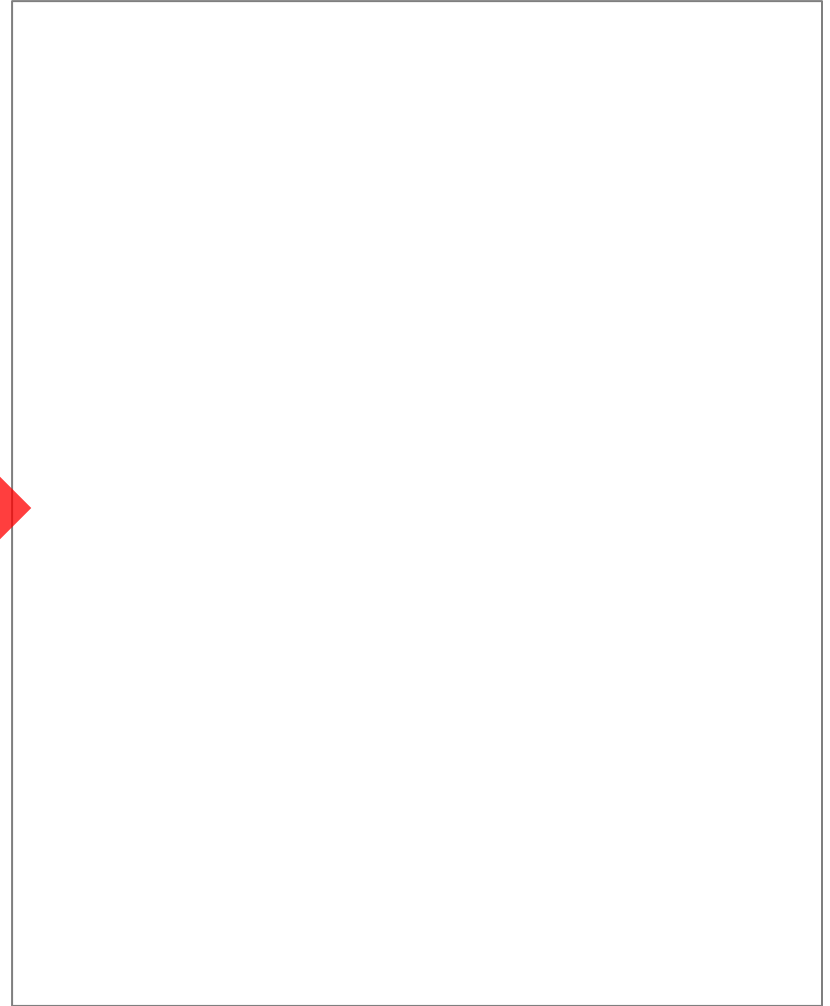
Exercise (solution)

Make a program that calculates the sum of the first ten numbers and leaves this value in register a0

$1 + 2 + 3 + ... + 10$
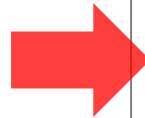
No valid ;-)

```
li  a0 0
add a0 a0 1
add a0 a0 2
add a0 a0 3
add a0 a0 4
add a0 a0 5
add a0 a0 6
add a0 a0 7
add a0 a0 8
add a0 a0 9
```

Exercise (solution)

Make a program that calculates the sum of the first ten numbers and leaves this value in register a0

```
int i, s;

s=0;
i=0;
while (i <= 10)
{
    s = s + i ;
    i = i + 1 ;
}
```
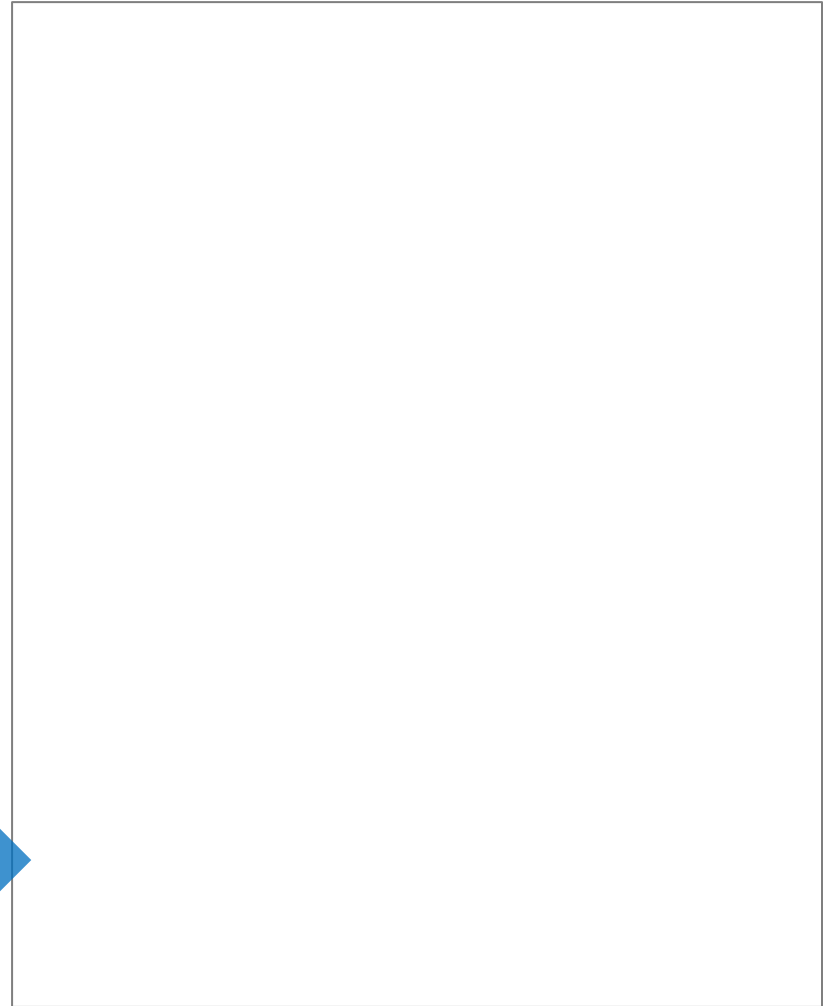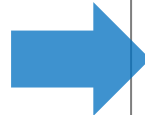
Exercise (solution)

Make a program that calculates the sum of the first ten numbers and leaves this value in register a0

```
int i, s;

s=0;
i=0;
while (i <= 10)
{
    s = s + i ;
    i = i + 1 ;
}
```

```
            li t0 0
            li a0 0
            li t2 10
while1:
            bgt  t0 t2 fin1
            add  a0 a0 t0
            addi t0 t0 1
            j    while1
fin1:
```

ARCOS @ UC3M

Félix García-Carballeira, Alejandro Calderón Mateos

Exercise

▸ Calculate the number of 1's of a register (t0).
  Result in t3.

ARCOS @ UC3M

Félix García-Carballeira,   Alejandro Calderón Mateos

Exercise (solution)

▸ Calculate the number of 1's of a register (t0).
Result in t3.

```
i = 0;
n = 45;   # number
s = 0;
while (i < 32)
{
  b = first bit of n
  s = s + b;
   shift the contents of
   of n one bit to the
   right
  i = i + 1 ;
}
```

ARCOS @ UC3M

Félix García-Carballeira,   Alejandro Calderón Mateos

Exercise (solution)

▸ Calculate the number of 1's of a register (t0).
Result in t3.

```
i = 0;
n = 45;   # number
s = 0;
while (i < 32)
{
  b = first bit of n
  s = s + b;
   shift the contents of
   of n one bit to the
   right
  i = i + 1 ;
}
```

```
i = 0;
n = 45;   # number
s = 0;
while (i < 32)
{
  b = n & 1;
  s = s + b;
  n = n >> 1;
  i = i + 1 ;
}
```

Exercise (solution)

▸ Calculate the number of 1's of a register (t0).
  Result in t3.

```
i = 0;
n = 45;   # number
s = 0;
while (i < 32)
{
  b = n & 1;
  s = s + b;
  n = n >> 1;
  i = i + 1 ;
}
```
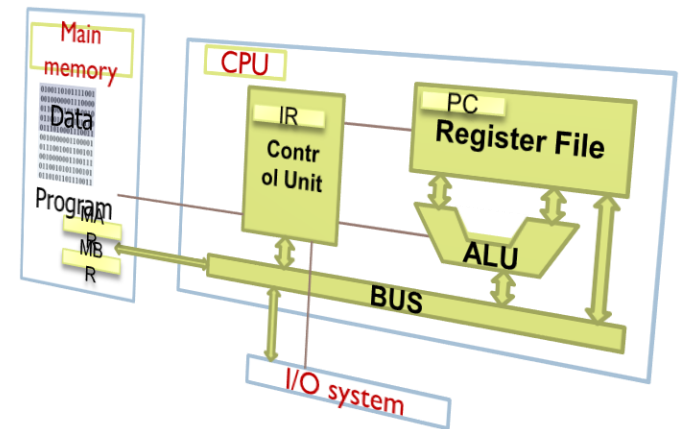
```
           li    t0, 0     #i
           li    t1, 45    #n
           li    t2, 32
           li    t3, 0     #s
while:     bge   t0, t2, fin
           andi  t4, t1, 1
           add   t3, t3, t4
           srli  t1, t1, 1
           addi  t0, t0, 1
           j     while
fin:       ...
```

Types of instructions
summary

- ▸ Data transfer
- ▸ Arithmetic
- ▸ Logical
- ▸ Shifting and rotation
- ▸ Comparison
- ▸ Control flow (branches, …)
- ▸ Conversion
- ▸ Input/output
- ▸ System calls

Typical faults

1) Poorly designed program
   - Does not do what is requested
   - Incorrectly does what is requested

2) Programming directly in assembler
   - Do not code in pseudo-code the algorithm to be implemented

3) Write unreadable code
   - Do not tabulate the code
   - Do not comment the assembly code or make reference to the algorithm initially proposed.

ARCOS @ UC3M

Félix García-Carballeira,   Alejandro Calderón Mateos

ARCOS Group

uc3m | Universidad **Carlos III** de Madrid

L3: Fundamentals of assembler programming
Computer Structure

Bachelor in Computer Science and Engineering

Bachelor in Applied Mathematics and Computing

Dual Bachelor in Computer Science and Engineering and Business Administration