ARCOS Group

uc3m | Universidad Carlos III de Madrid

Lesson 3 (I)
Fundamentals of assembler programming

Computer Structure

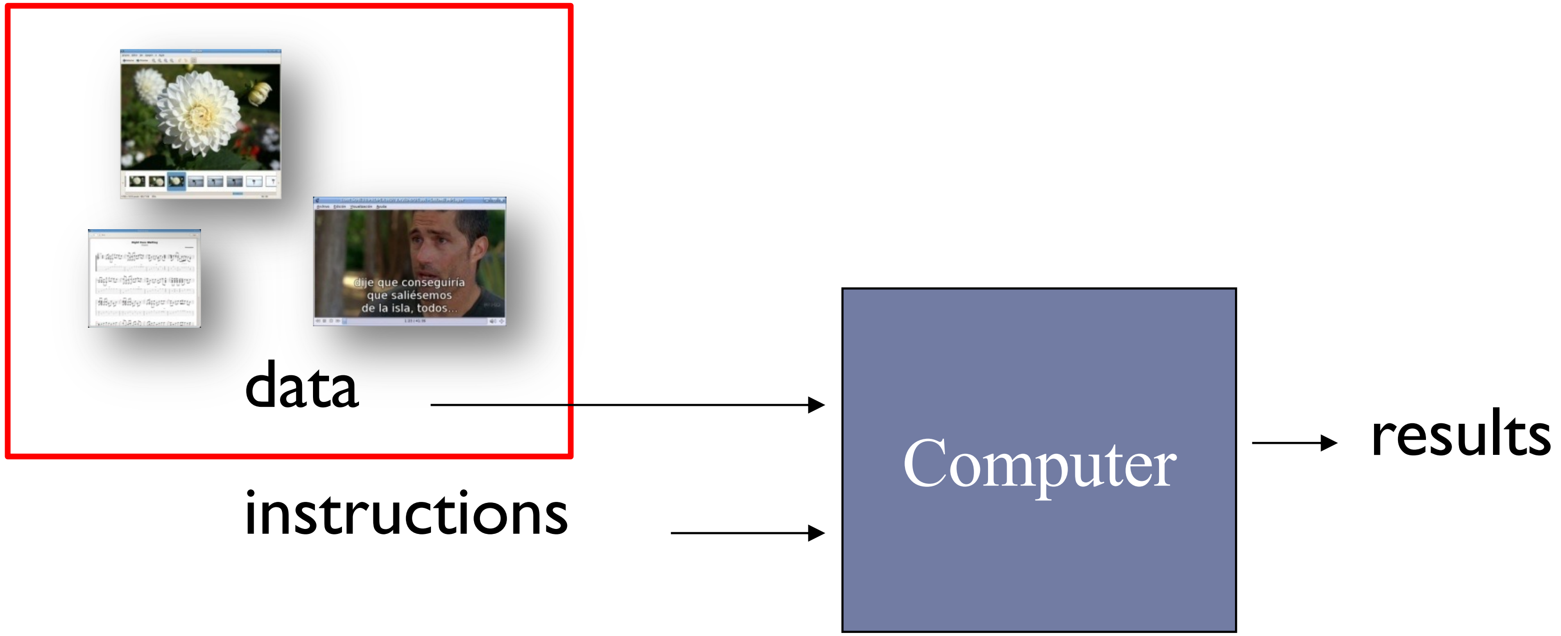
Bachelor in Computer Science and Engineering

Contents

Félix García-Carballeira, Alejandro Calderón Mateos

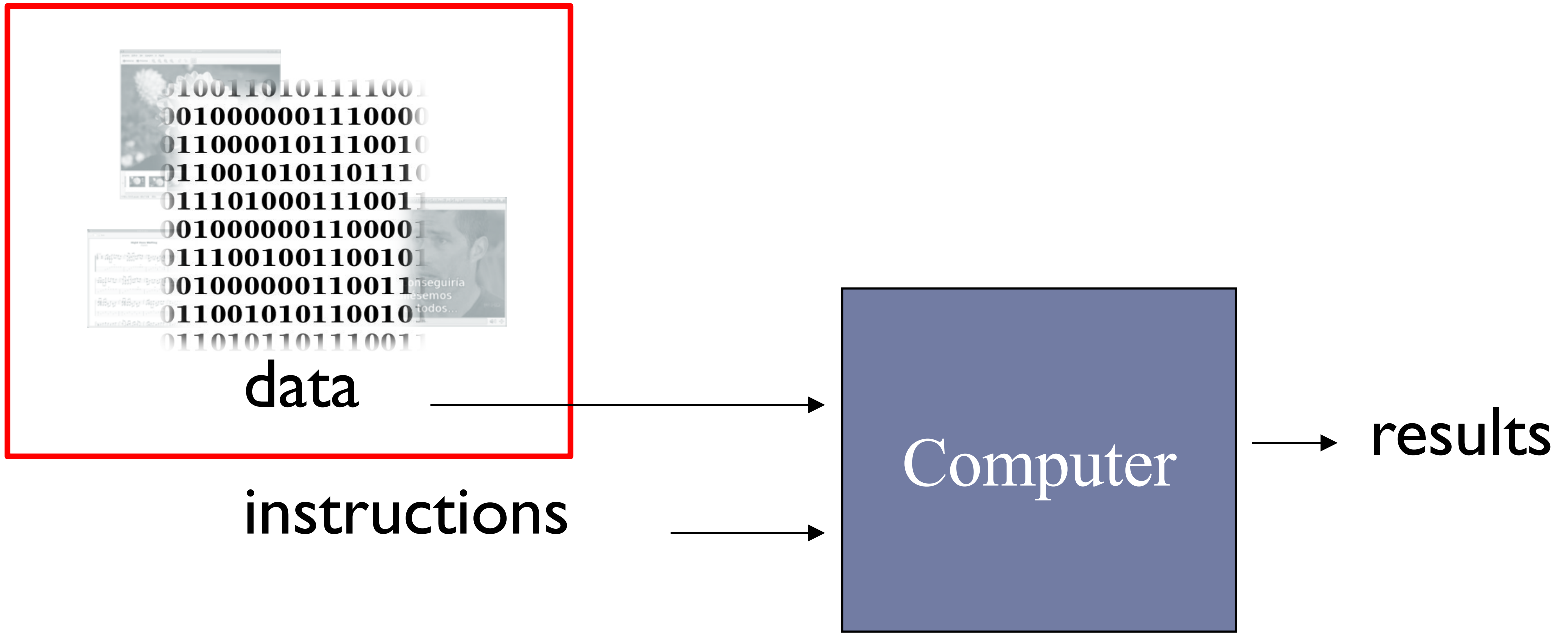
Types of information: instructions and data

- Data representation…



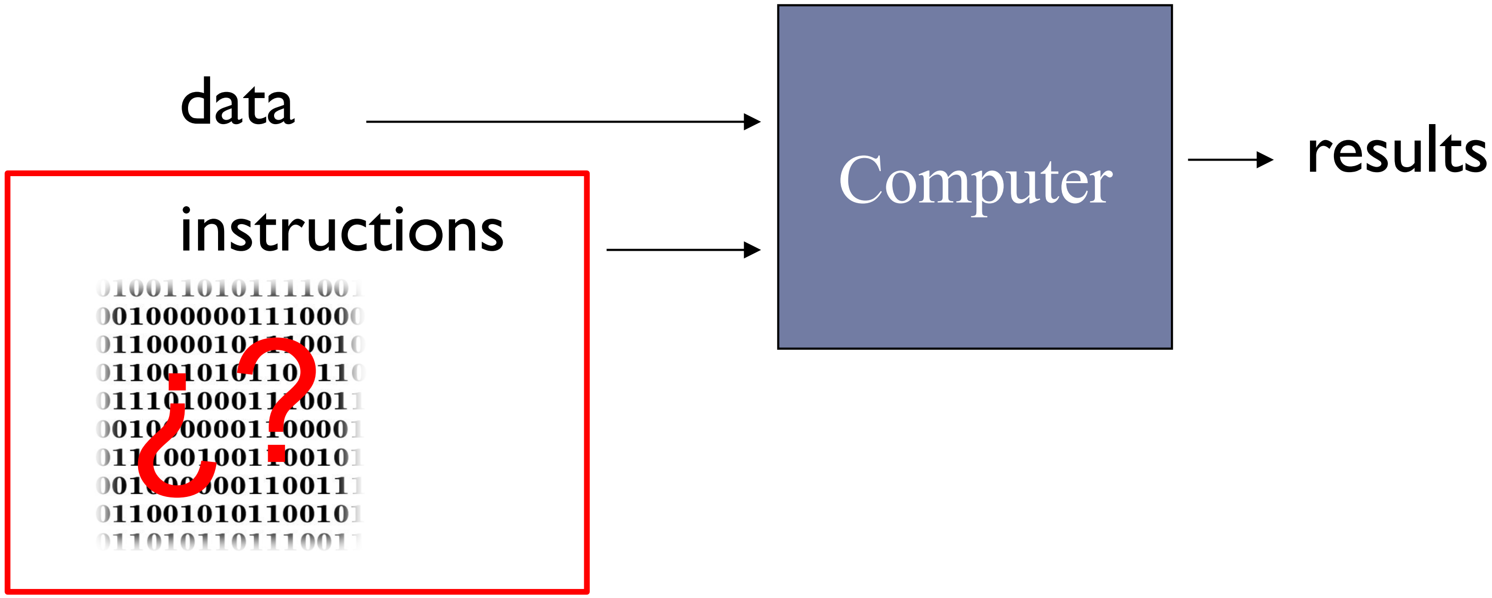
Types of information: instructions and data

- Binary data representation.



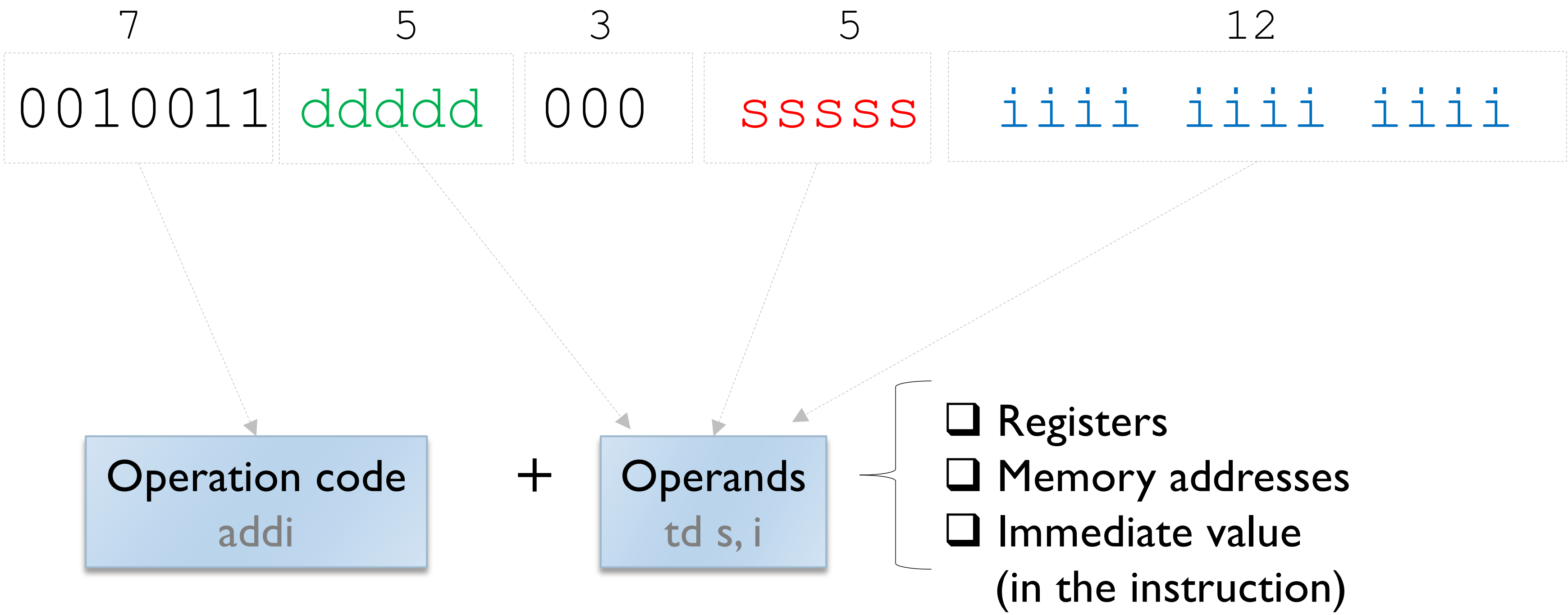
Types of information: instructions and data

- What about the instructions?



Machine instruction

- Machine instruction: elementary operation that can be executed directly by the processor.
- Example of instruction in RISC-V:
  - Sum of a register (s) with an immediate value (i) and the result of the sum is stored in register (t).



Properties of machine instructions

- 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)

Information contained in a machine instruction

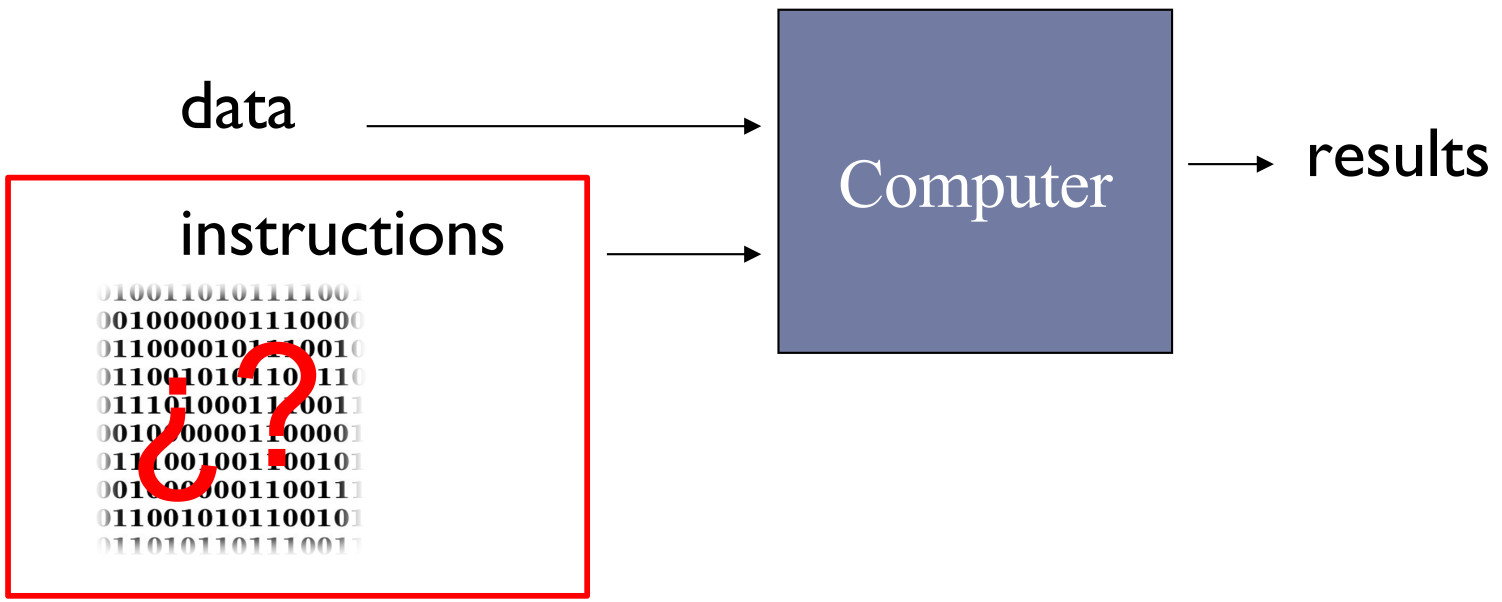
- The operation to be performed.
- Where the operands are located:
  - In registers
  - In memory
  - In the instruction itself (immediate)
- Where to leave the results (as operand)
- A reference to the next instruction to be executed
  - Implicitly: the following instruction
    - A program is a consecutive sequence of machine instructions.
  - Explicitly in branching instructions (as operand)

Operation code	+	Operands
addi		d, s, i

Operands:
- ❑ Registers
- ❑ Memory addresses
- ❑ Immediate value (in the instruction)

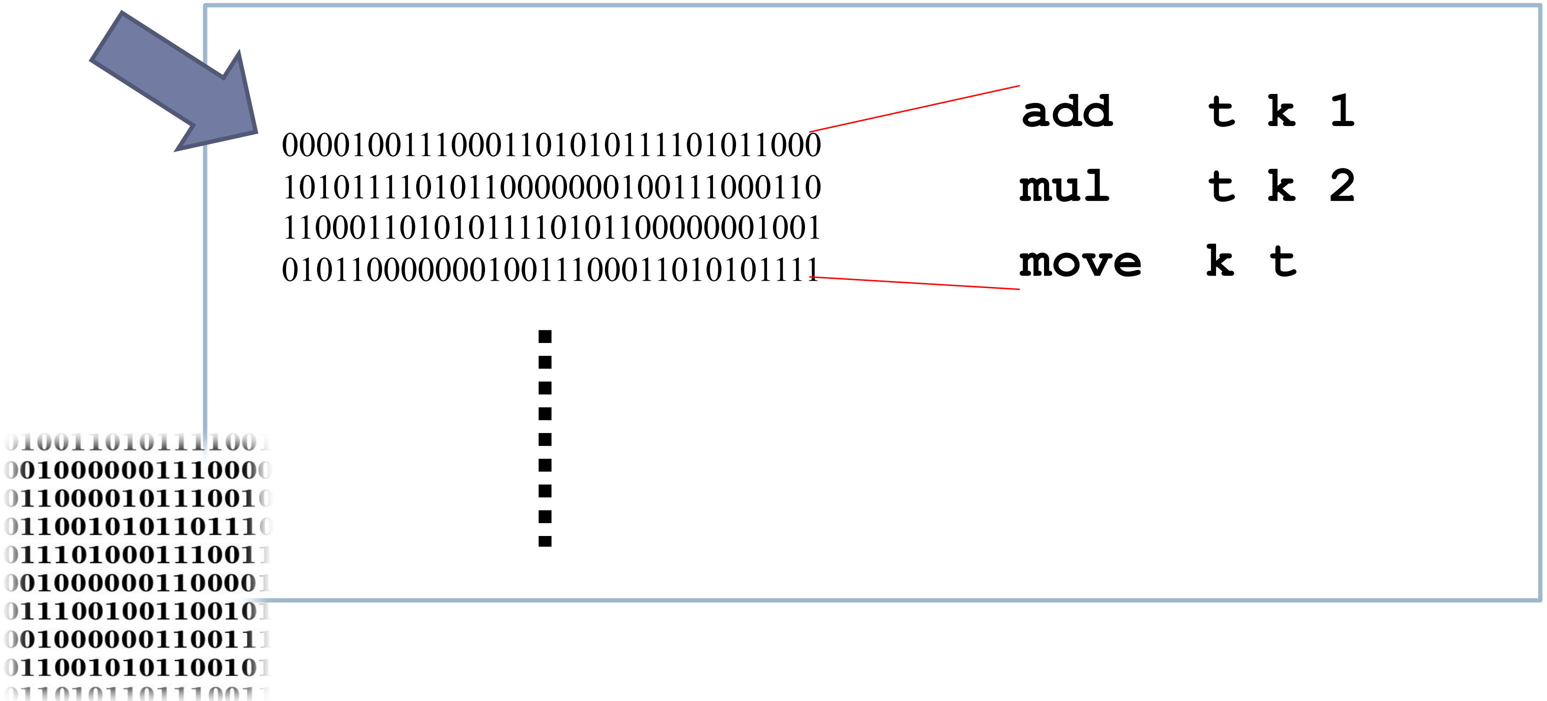
Types of information: instructions and data

- What about the instructions?



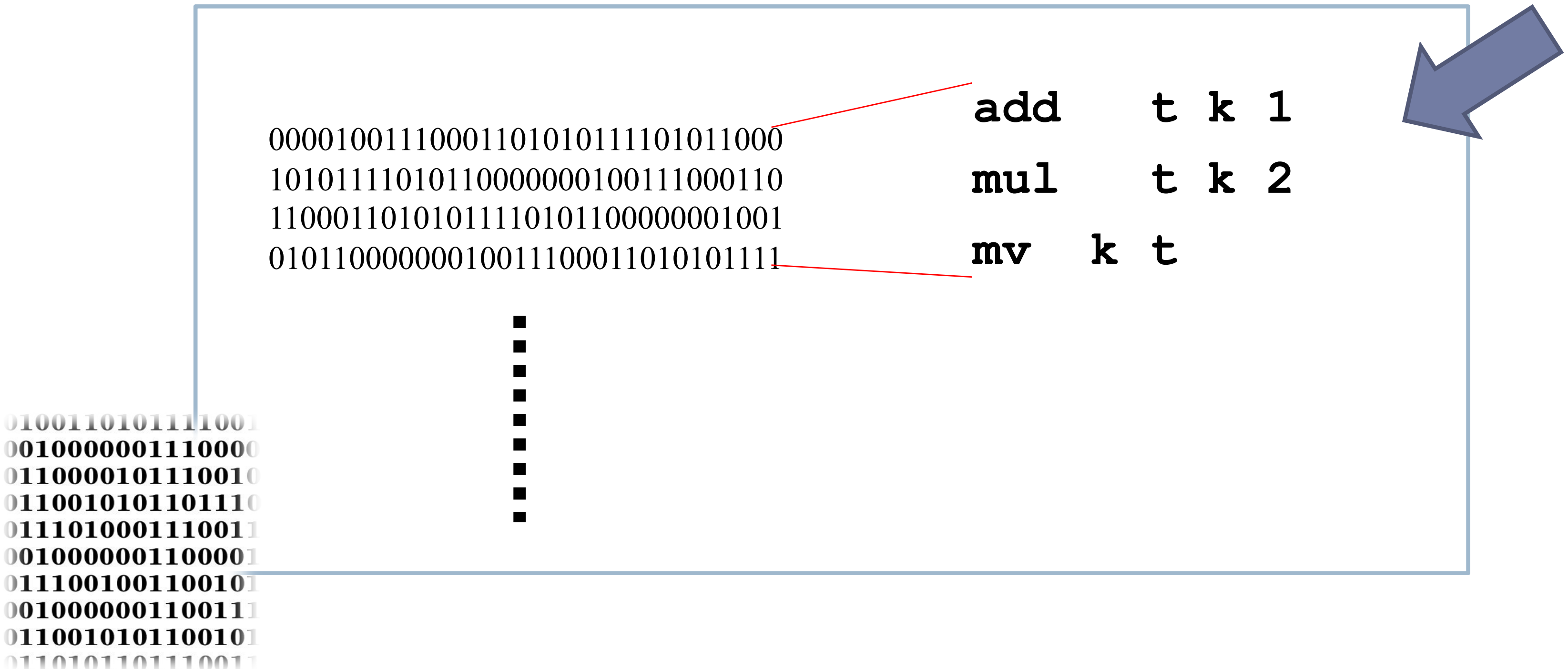
Definition of program

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



Assembly language definition

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

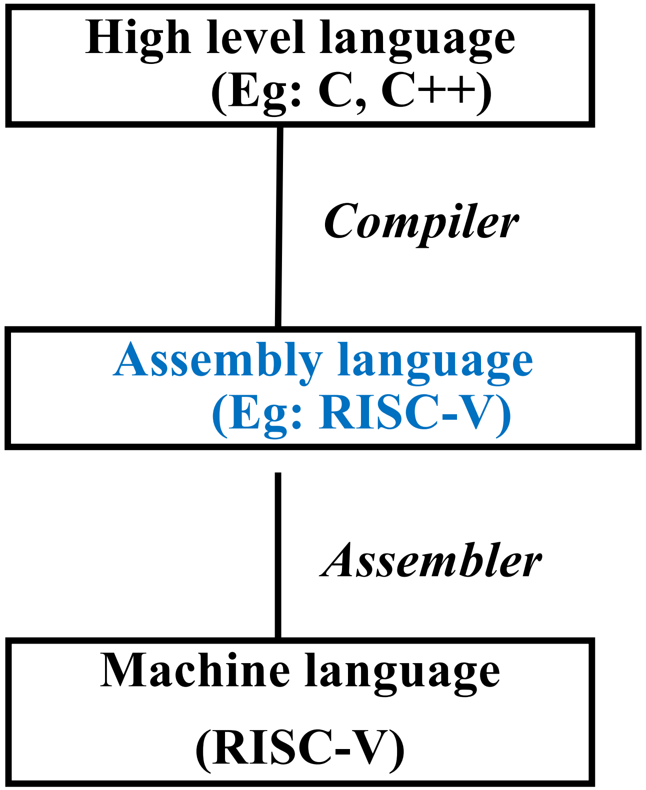


Félix García-Carballeira, Alejandro Calderón Mateos

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




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

```
lw   t0, 0(x2)
lw   t1, 4(x2)
sw   t1, 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)

- Operands:
  - Registers, memory, the instruction itself
- Memory addressing:
  - Most of them use byte addressing
  - They provide instructions for accessing multi-byte elements from a given position
- Addressing modes:
  - They specify where and how to access operands (register, memory or the instruction itself)
- Type and size of operands:
  - bytes: 8 bits
  - integers: 16, 32, 64 bits
  - floating-point numbers: single precision, double precision, etc.

Characteristics of an instruction set (2/2)

- Operations:
  - Arithmetic, logic, transfer, control, control, etc.
- Flow control instructions:
  - Unconditional jumps
  - Conditional jumps
  - Procedure calls
- Format and coding of the instruction set:
  - Fixed or variable length instructions
    - 80x86: variable (from 1 up to 18 bytes)
    - RISC-V, ARM: fixed

Programming model of a computer

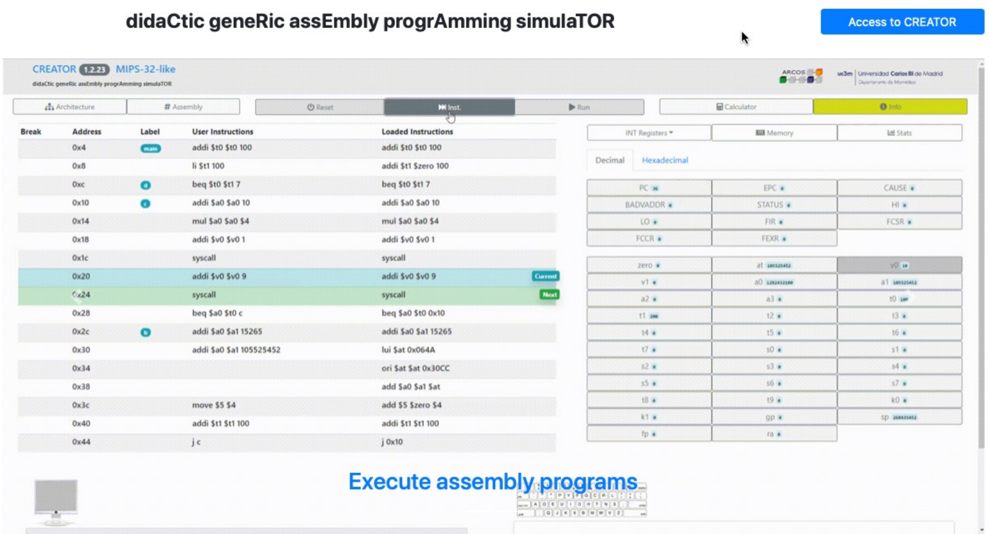
- A computer offers a programming model that consists of:
  - Instruction set (assembly language)
    - ISA: *Instruction Set Architecture*
    - An instruction includes:
      - Operation code
      - Other elements: registers, memory address, numbers
  - Storing elements
    - Registers
    - Memory
    - Registers of I/O controllers
  - Execution modes

Motivation to learn assembly

```c
#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++, Java, …
- Analyze the execution time of high-level instructions.
- Useful in specific domains:
  - Compilers
  - Operating Systems
  - Games
  - Embedded systems
  - Etc.

Motivation to use CREATOR simulator

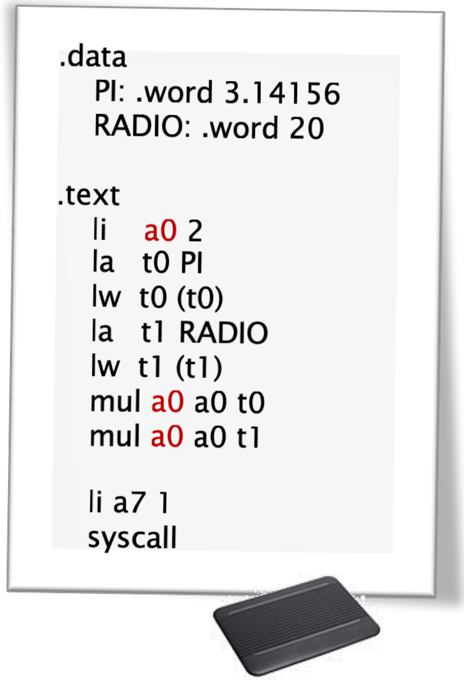


https://creatorsim.github.io/

- CREATOR: didaCtic geneRic assEmbly progrAmming simulaTOR
- CREATOR can simulate RISC-V32 and RISC-V architectures
- CREATOR can be executed from Firefox, Chrome or Edge

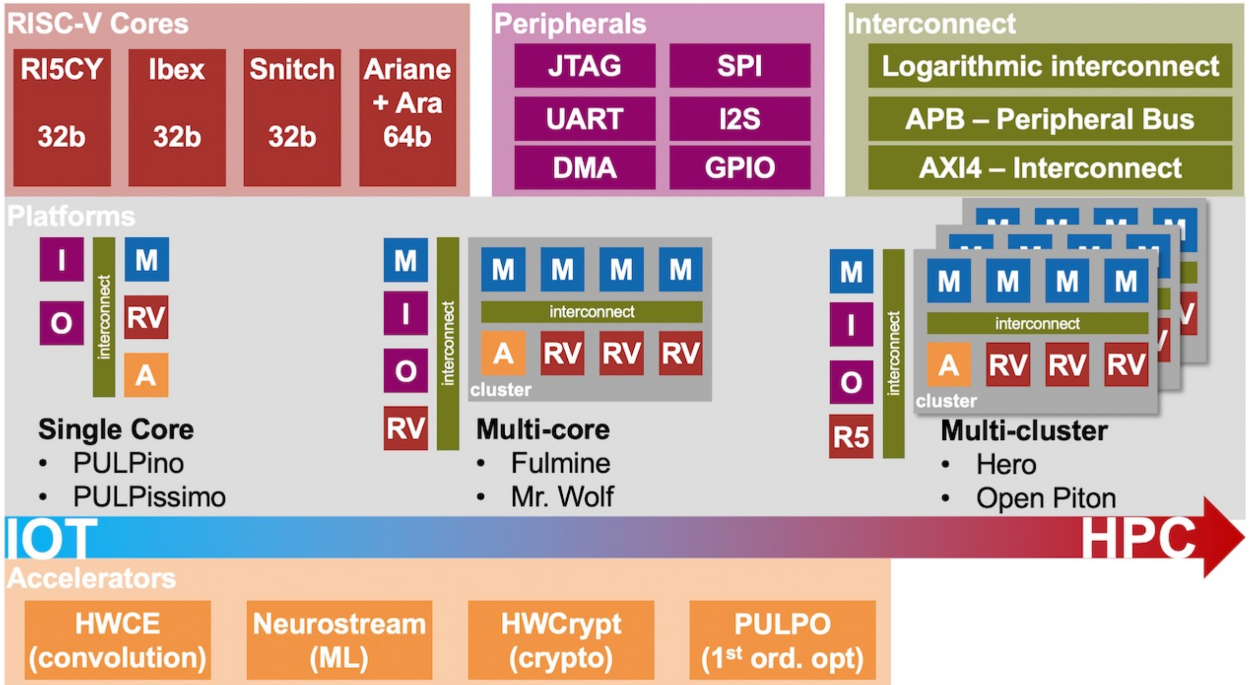
Goals

- 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



Example for assembly: RISC-V

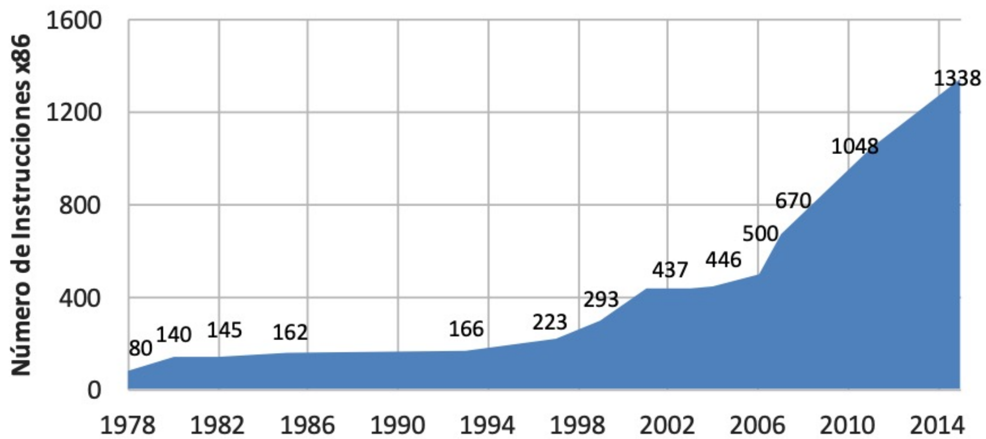
- RISC (*Reduced Instruction Set Computer*) processor.
- Examples of RISC processors:
  - RISC-V, ARM, MIPS, etc.



https://pulp-platform.org/img/main/pulp_family.png

Benefits of using RISC-V

- Open hardware architecture:
  - Allows anyone to design, manufacture and sell RISC-V chips and software.
- Small and simple instruction set
- Difference with x86 architecture instructions



Guía Práctica de RISC-V.
David Patterson and Andrew Waterman

Contents

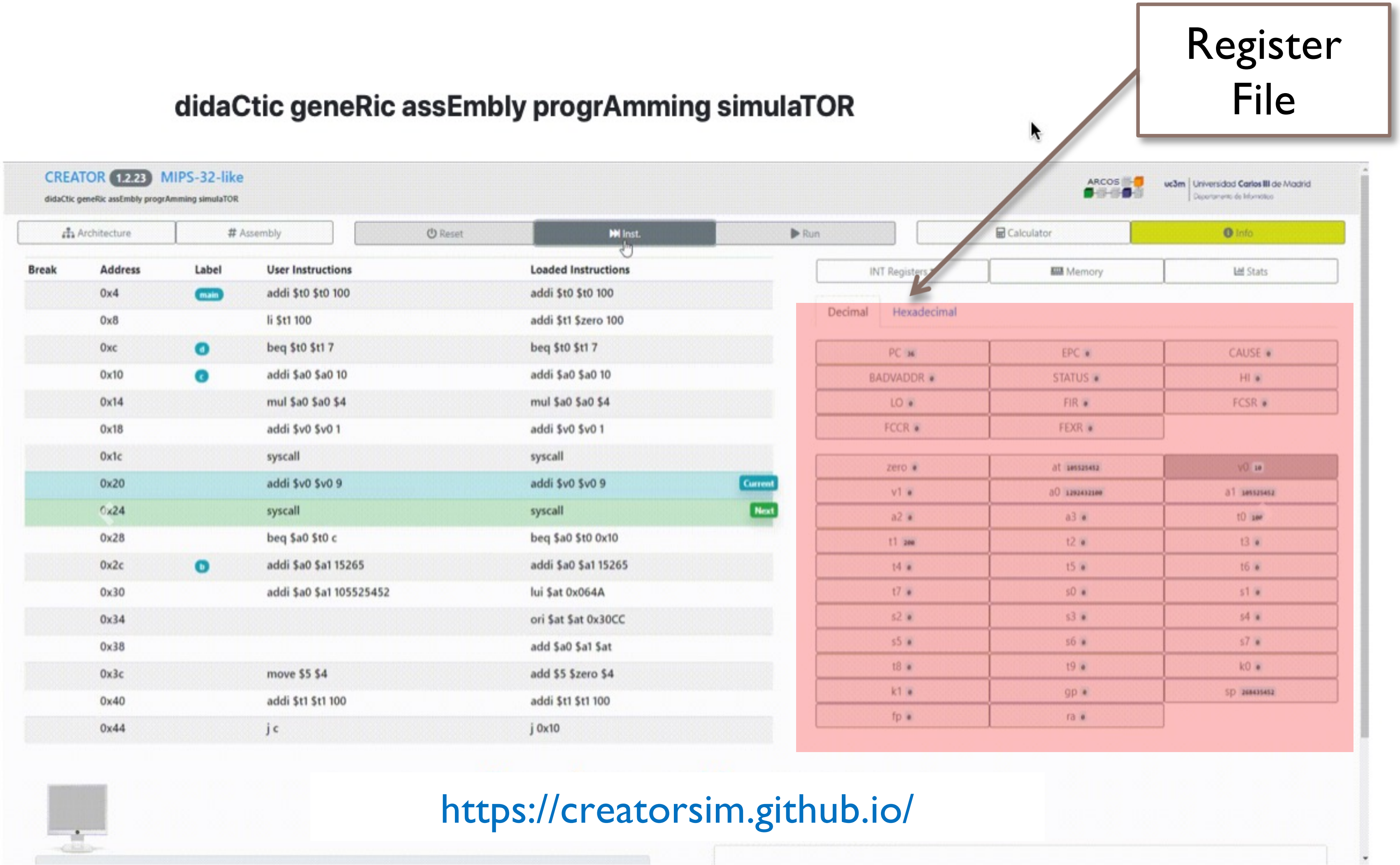
RISC-V instruction set

- Modular instruction set:
  - RV32I: integer instruction set. 32 bits
  - RV64I: integer instruction set. 64 bits
  - RV128I: integer instruction set. 128 bits
- Each one has different extensions:
  - M: instructions for integer multiplication and division
  - F: single-precision floating-point instructions
  - D: double-precision floating-point instructions
  - G: Includes M, F and D
  - Q: quadruple-precision floating-point instructions
  - Etc.
- Example: RV64F -> 64-bit RISC-V processor with single-precision floating-point instructions

RISC-V instructions used in the course

- Modular instruction set:
  - RV32I: integer instruction set. 32 bits
  - RV64I: integer instruction set. 64 bits
  - RV128I: integer instruction set. 128 bits
- Each one has different extensions:
  - M: instructions for integer multiplication and division
  - F: single-precision floating-point instructions
  - D: double-precision floating-point instructions

CREATOR



Félix García-Carballeira, Alejandro Calderón Mateos

Register File (integers)

Register	ABI Name
x0	zero
x1	ra
x2	sp
x3	gp
x4	tp
x5	t0
x6–7	t1–2
x8	s0/fp
x9	s1
x10–11	a0–1
x12–17	a2–7
x18–27	s2–11
x28–31	t3–6

ABI: application binary interface

Register File (integers)

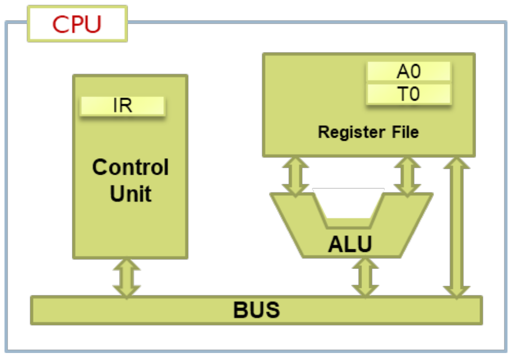
Symbolic name	Number	Usage
zero	x0	Constant 0
ra	x1	Return address (routines/functions)
sp	x2	Stack pointer
gp	x3	Global pointer
tp	x4	Thread pointer
t0...t2	x5-x7	Temporary (NO preserved across calls)
s0/fp	x8	Saved temporary (preserved across calls) / Frame pointer
s1	x9	Saved temporary (preserved across calls)
a0...a1	x10…11	Arguments for routines/return value
a2...a7	12…x17	Arguments for routines
s2… s11	x18…x27	Saved temporary (preserved across calls)
t3...t6	x28…x31	Temporary (NO preserved across calls)

- There are 32 registers
  - Size: 4 bytes (1 word)
  - Double naming: logical and numerical (with x at the beginning)
- Use convention
  - Reserved
  - Arguments
  - Results
  - Temporary
  - Pointers

Data transfer

- Copy data:
  - Between registers
  - Between registers and memory (later)

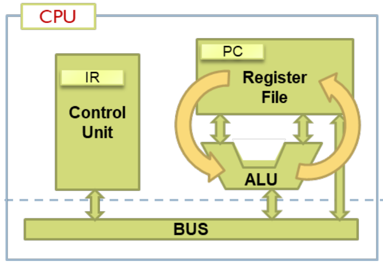
- Examples:
  - Immediate load
    - li t0 5   # t0 ← 5
  - Register to register
    - mv a0 t0   # a0 ← t0




```
move  a0  t0     # a0      t0
li        t0  5  # t0      000....00101
```


Félix García-Carballeira, Alejandro Calderón Mateos

Arithmetic instructions



- Integer operations (ALU) or floating point operations (FPU)
- 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
  - Division
    div t0, t1, t2   t0 ← t1 / t2 Integer division
    rem t0, t1, t2   t0 ← t1 % t2 remainder

Example

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

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

Example

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

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

```
li t0, 5
li t1, 7
li t2, 8


add t1, t1, t2
mul t3, t1, t0
```

Example

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


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

Example

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

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

```
li t0, 5
li t1, 7
li t2, 8
li t3 10

sub t4, t1, t3
mul t4, t4, t0
add t4, t4, t2
li  t5, -1
mul t4, t4, t5
```

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
```

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
```

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

addi  t4 t2 -10
mul   t4 t4 t1
add   t4 t4 t3
mul   t4 t4 -1
```

Register File (floating point)

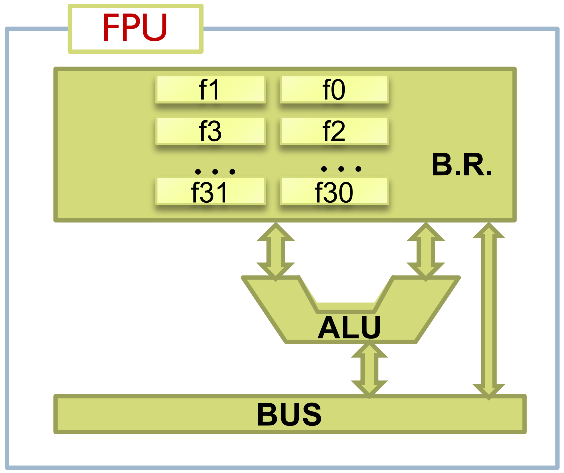
Symbolic name	Numbered name	Uso
ft0-ft7	f0 … f7	Temporals (like t…)
fs0-fs1	f8 … f9	Saved (like s…)
fa0-fa1	f10 … f11	Arguments/return (like a0/a1)
fa2-fa7	f12 … f17	Arguments (like a…)
fs2-fs11	f18 … f27	Saved (like s…)
ft8-ft11	f28 … f31	Temporals (like t…)

- There are 32 registers
- For simple precision register are 4 bytes
- For double precision registers are 8 bytes
  - For single precision, values are stored in the less significant bits
  - For double precision are stored in all bits of the register

Arithmetic: IEEE 754

- IEEE 754 floating point arithmetic on the FPU

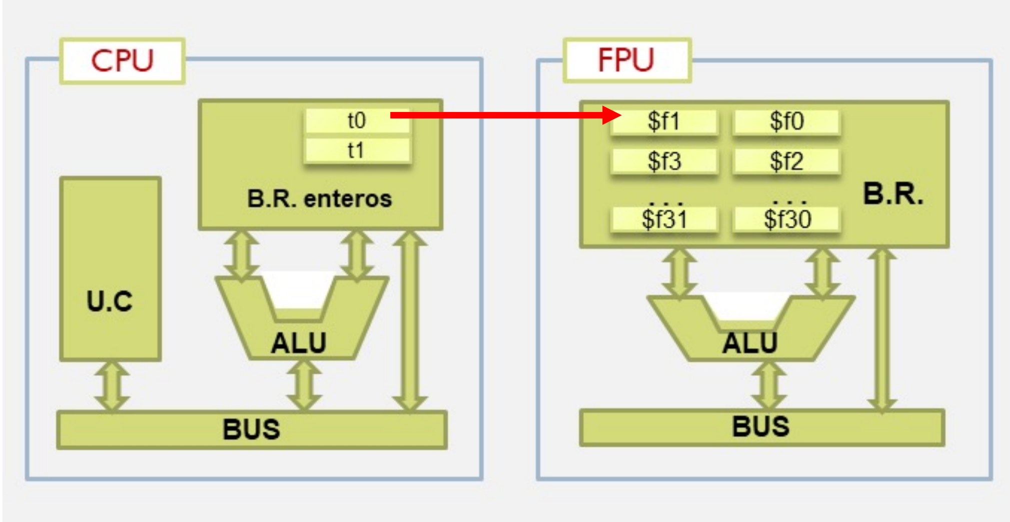
- Examples of common instructions:
  - fmv.s rd rs # rd = rs
  - fadd.s rd rs1 rs2 # rd = rs1 + rs2
  - fsub.s rd rs1 rs2 # rd = rs1 - rs2
  - fmul.s rd rs1 rs2 # rd = rs1 * rs2
  - fdiv.s rd rs1 rs2 # rd = rs1 / rs2
  - fmin.s rd rs1 rs2 # rd = min(rs1, rs2)
  - fmax.s rd rs1 rs2 # rd = max(rs1, rs2)
  - 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



Copy (Integer register<-> Floating point registers)

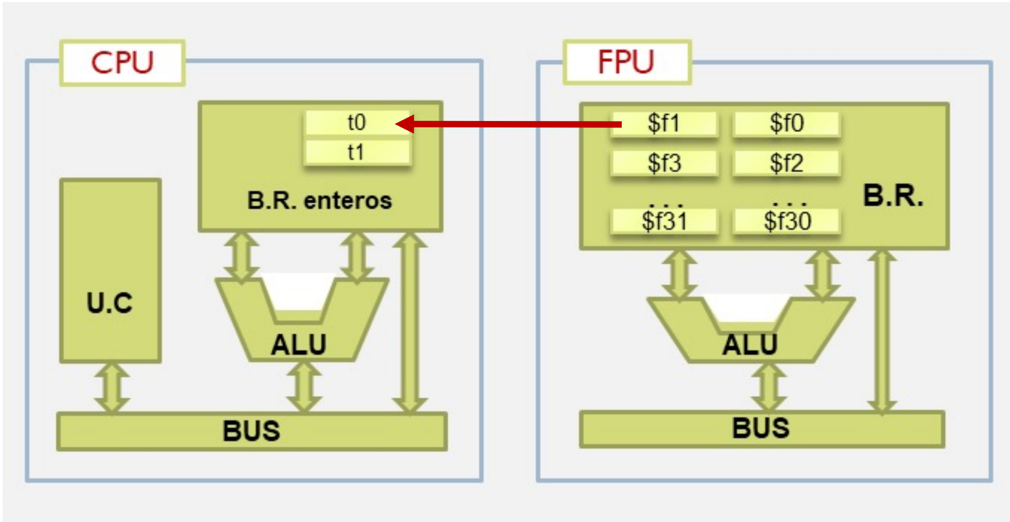
fmv.w.x rd rs

- Copy from integer rs to rd (*single precisión*)



fmv.x.w rd rs

- Copy from register rs (*single precision*) to integer register rd



Conversion operations (1/3)

integer <-> single precision

- `fcvt.w.s  rd, rs1`
  - Convert from single precision (value in floating register rs1) to 32-bits integer **with** sign (integer register rd).
- `fcvt.wu.s rd, rs1`
  - Convert from single precision (value in floating register rs1) to 32-bits integer **without** sign (integer register rd).
- `fcvt.s.w  rd, rs1`
  - Convert from 32-bits integer **with** sign (value in integer register rs1) to single precision (floating register rd).
- `fcvt.s.wu rd, rs1`
  - Convert from 32-bits integer **without** sign (value in integer register rs1) to single precision (floating register rd).

Conversion operations (2/3)

integer <-> double precision

- `fcvt.w.d  rd, rs1`
  - Convert from double precision (value in floating register rs1) to 32-bits integer **with** sign (integer register rd).
- `fcvt.wu.d rd, rs1`
  - Convert from double precision (value in floating register rs1) to 32-bits integer **without** sign (integer register rd).
- `fcvt.d.w  rd, rs1`
  - Convert from 32-bits integer **with** sign (value in integer register rs1) to single precision (floating register rd).
- `fcvt.d.wu rd, rs1`
  - Convert from 32-bits integer **without** sign (value in integer register rs1) to single precision (floating register rd).

Conversion operations (3/3)

double precision <-> simple precision

- `fcvt.s.d rd, rs1`
  - Convert from double precision (value in floating register rs1) to single precisión (floating register rd).
- `fcvt.d.s rd, rs1`
  - Convert from single precision (value in floating register rs1) to double precisión (floating register rd).

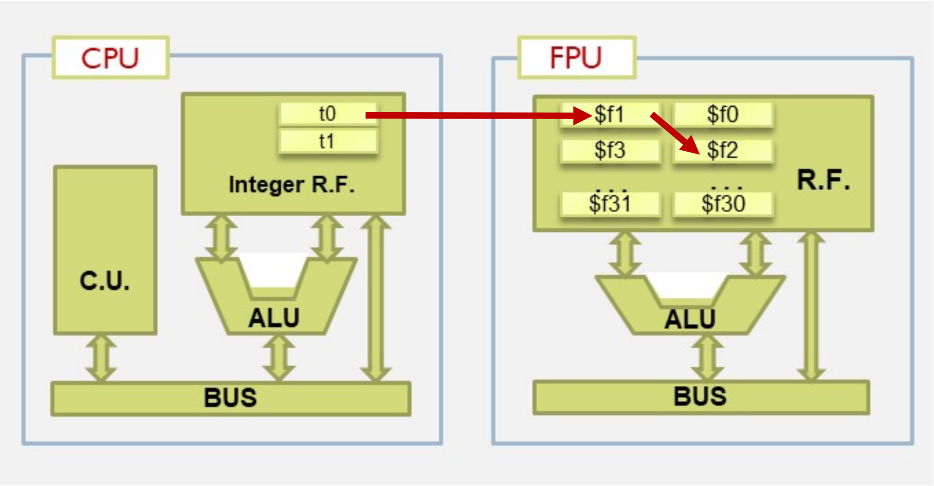
Floating point clasification

- fclass.s rd, rs1, rs2 (single precision)
- fcalss.d rd, rs1, rs2 (double precision)
- Write to rd the floating point number type of the rs1 register. :

Value en rd	Meaning
0	-Inf
1	Normalized negative
2	Not normalized negative
3	-0
4	+0
5	Normalized positive
6	Not normalized positive
7	+Inf
8	NaN
9	NaN

Example

```c
float PI     = 3,1415;
int   radio = 4;
float length;

length = PI * radio;
```




```
.text

main:
    # no “li.s” instruction
    # 0x40490E56 represents
    # 3.1415 in hexadecimal

    li         t0,  0x40490E56
    fmv.w.x    ft0, t0     # ft0 ←t0
    li         t0   4       # 4 en Ca2
    fcvt.s.w   ft1, t0      # 4 ieee754
    fmul.s     ft0, ft0, ft1
```

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

Example

```
li t0, 5
li t1, 8

and t2, t1, t0
```

What is the value of t2?

Solution

```
li t0, 5
li t1, 8

and t2, t1, t0
```

What is the value of t2?

	000 …. 0101	t0
	000 ….. 1000	t1
and	000 ….. 0000	t2

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
```

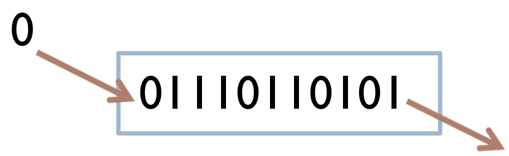
What does an "and" with 0x007FFFFF allow to do?

Obtain the 23 least significant bits

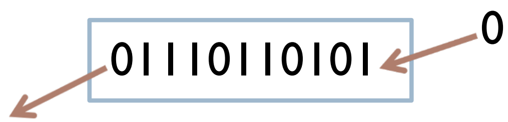
The constant used for bit selection is called a mask.

Shift instructions

- Bits movement
- Examples:
  - Shift right logical
    srli t0 t0 4   (t0 = t0 >> 4 bits)



  - Shift left logical
    slli t0 t0 5   (t0 = t0 << 5 bits)



  - Shift right arithmetic
    srai t0 t0 2   (t0 = t0 >> 2 bits)

Example (solution)

```
li t0, 5
li t1, 6

srai t0, t1, 1
```

- What is the value of t0?

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

```
slli t0, t1, 1
```

- What is the value of t0?

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

Ejercicio

```
Write 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.
```

Ejercicio (solución)

```
Write 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)

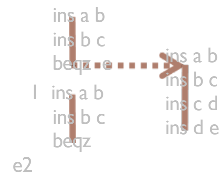
- Single precision
  - feq.s rd, rs1, rs2 — if (rs1== rs2) rd= 1; else rd = 0
  - fle.s rd, rs1, rs2 — if (rs1<= rs2) rd= 1; else rd = 0
  - flt.s rd, rs1, rs2 — if (rs1< rs2) rd= 1; else rd = 0

- Double precicion:
  - 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

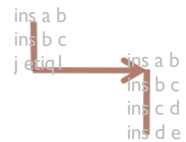
Branch instructions

- Change the sequence of instructions to be executed

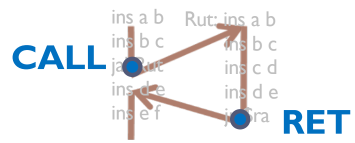
- Several types:
  - Conditional branches:
    - Branch if value match condition
    - E.g.: bne t0 t1 etiqueta1



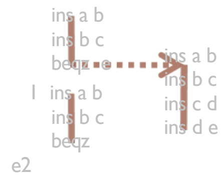
  - Unconditional branches:
    - Always branch
      E.g.: j etiqueta2



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



Branch instructions

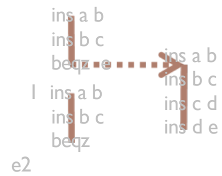


- Change the sequence of instructions to be executed

- Conditional branches:

```
▸ beq    t0   t1   etiq    # salta a etiq1 si t0 == t1
▸ bne    t0   t1   etiq    # salta a etiq1 si t0 != t1
▸ blt    t0   t1   etiq    # salta a etiq1 si t0 <  t1
▸ bltu   t0   t1   etiq    # salta a etiq1 si t0 <  t1 (unsigned)
▸ bge    t0   t1   etiq    # salta a etiq1 si t0 >= t1
▸ bgeu   t0   t1   etiq    # salta a etiq1 si t0 >= t1 (unsigned)

(as pseudoinstructions)
▸ bgt    t0   t1   etiq    # salta a etiq1 si t0 >  t1
▸ ble    t0   t1   etiq    # salta a etiq1 si t0 <= t1
```

Branch instructions



- Change the sequence of instructions to be executed
- Conditional branches:
```
beq     t0   t1   etiq    # salta a etiq1 si t0 == t1
bne     t0   t1   etiq    # salta a etiq1 si t0 != t1
blt     t0   t1   etiq    # salta a etiq1 si t0 <  t1
bltu    t0   t1   etiq    # salta a etiq1 si t0 <  t1 (unsigned)
bge     t0   t1   etiq    # salta a etiq1 si t0 >= t1
bgeu    t0   t1   etiq    # salta a etiq1 si t0 >= t1 (unsigned)
bgt     t0   t1   etiq    # salta a etiq1 si t0 >  t1
ble     t0   t1   etiq    # salta a etiq1 si t0 <= t1
```
- Incondicional:
```
j       etiq              # salta a etiq
```

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

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


ARCOS @ UC3M
Félix García-Carballeira, Alejandro Calderón Mateos

Control flow structures
if

```
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;
  }
  ...
}
```

Control flow structures if...(1/2)

```
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:    ...
```

Estructuras de control
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;
  }
  ...
}
```

```
        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 structures
if-else


```
int a=1;
int b=2;

main ()
{
  if (a < b){
      // acción 1
  } else {
      // acción 2
  }
}
```

```
 li   t1 1
        li   t2 2

if_3:   bge t1 t2 else_3

then_3: # acción 1
        j   fi_3

else_3: # acción 2

fi_3: ...
```

Exercise

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

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

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:   ...
```

Branchs with floating point register

Jump to etiq if:

ft1 < ft2

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

Control flow structures while

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

```
int i;

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

     /* action */
     i = i + 1 ;
   }
}
```

Control flow structures 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 ;
    }
}
```

```
          li   t0 0
          li   t1 10

while2:   bge   t0   t1   end2
          # action
          addi t0 t0 1
          j     while2
end2:      ...
```

Exercise

- Calculate 1 + 2 + 3 + …. + 10 and result in a0

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


Félix García-Carballeira, Alejandro Calderón Mateos

Exercise (solution)

- Calculate 1 + 2 + 3 + …. + 10 and result in a0

```
i=0;
s=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
        add t0 t0 1
        j   while1
fin1:
```

Exercise

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

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

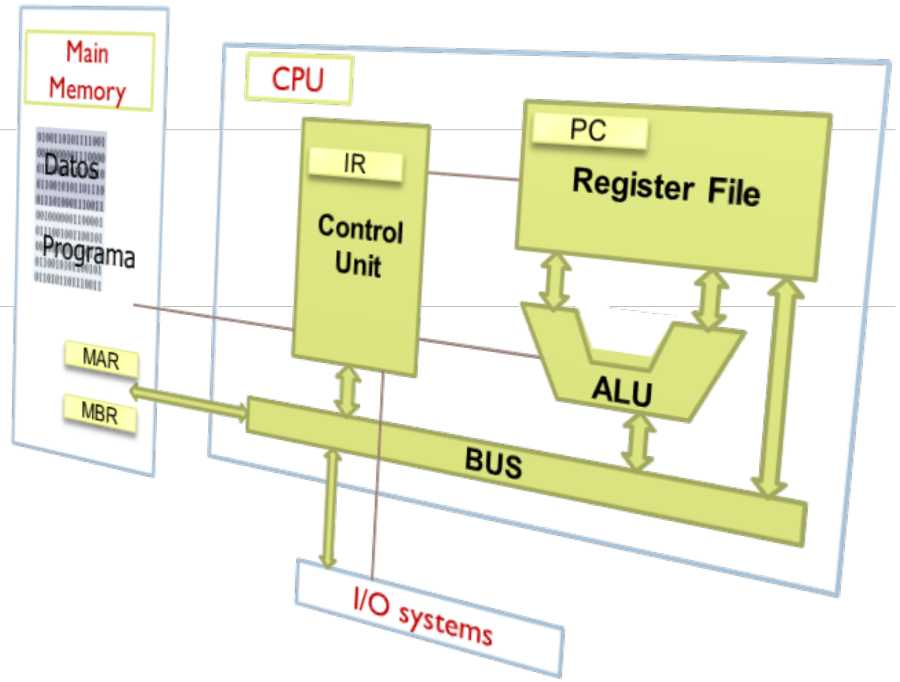
Exercise (solution)

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

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

```
             li    t0, 0     #i
             li    t1, 45    #n
             li    t2, 32
            li    t3, 0     #s
while:      bge   t0, t2, end
            andi  t4, t1, 1
            add   t3, t3, t4
            srli  t1, t1, 1
            addi  t0, t0, 1
            j      while
end:        ...
```

Types of instructions
summary



- Data transfer
- Arithmetic
- Logical
- Shifting
- Comparison
- 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.

Example

- Calculate the number of 1's of a `int` in C/Java

Another solution :

```
int count[256] = {0,1,1,2,1,2,2,3,1, . . .8};
int i;
int c = 0;

for (i = 0; i <4; i++) {
     c =  count[n & 0xFF];
     s = s + c;
     n = n >> 8;
}
printf("There is  %d\n", c);
```

Example

- Obtain the 16 first bits of a register (t0) and store them in the 16 last bits of other register (t1)

Solution

- Obtain the 16 first bits of a register (t0) and store them in the 16 last bits of other register (t1)

```
srl   t1,   t0,  16
```

Shift 16 bits to right

Example

- Determine if the number stored in t2 is even. If t2 is even the program stores 1 in t1, else stores 0 in t1

Félix García-Carballeira, Alejandro Calderón Mateos

Solution

- Determine if the number stored in t2 is even. If t2 is even the program stores 1 in t1, else stores 0 in t1

```
        li    t2   9
        li    t1   2
        rem   t1   t2   t1        # remainder
        beq   t1   x0   then      # cond.
else:   li    t1  0
        j      end          # uncond.
then:   li t1 1
end:  ...
```

Example

- Determine if the number stored in t2 is even. If t2 is even the program stores 1 in t1, else stores 0 in t1. In this case, analyze the last bit

Solution

- Determine if the number stored in t2 is even. If t2 is even the program stores 1 in t1, else stores 0 in t1. In this case, analyze the last bit

```
        li    t2  9
        li    t1  1
        and   t1  t2  $t1     # get the last bit
        beq   t1  x0  then    # cond.
else:   li    t1  0
        j     end         # uncond.
then:   li    t1  1
end:    ...
```

Example

- Calculate a^n
  - a in t0
  - n in t1
  - Result in a0

```
a=8
n=4;
i=0;
p = 1;
while (i < n)
{
  p = p * a
  i = i + 1 ;
}
```

Solution

- Calculate a^n
  - a in t0
  - n in t1
  - Result in a0

```
a=8
n=4;
i=0;
p = 1;
while (i < n)
{
   p = p * a
   i = i + 1 ;
}
```

```
        li    t0, 8
        li    t1, 4
        li    t2, 1
        li    t4, 0

while:  bge   t4, t1, fin
        mul   t2, t2, t0
        addi  t4, t4, 1
        j     while
fin:    move  a0,  t2
```