Computer Architecture (Practical Class)
Introduction to Assembly

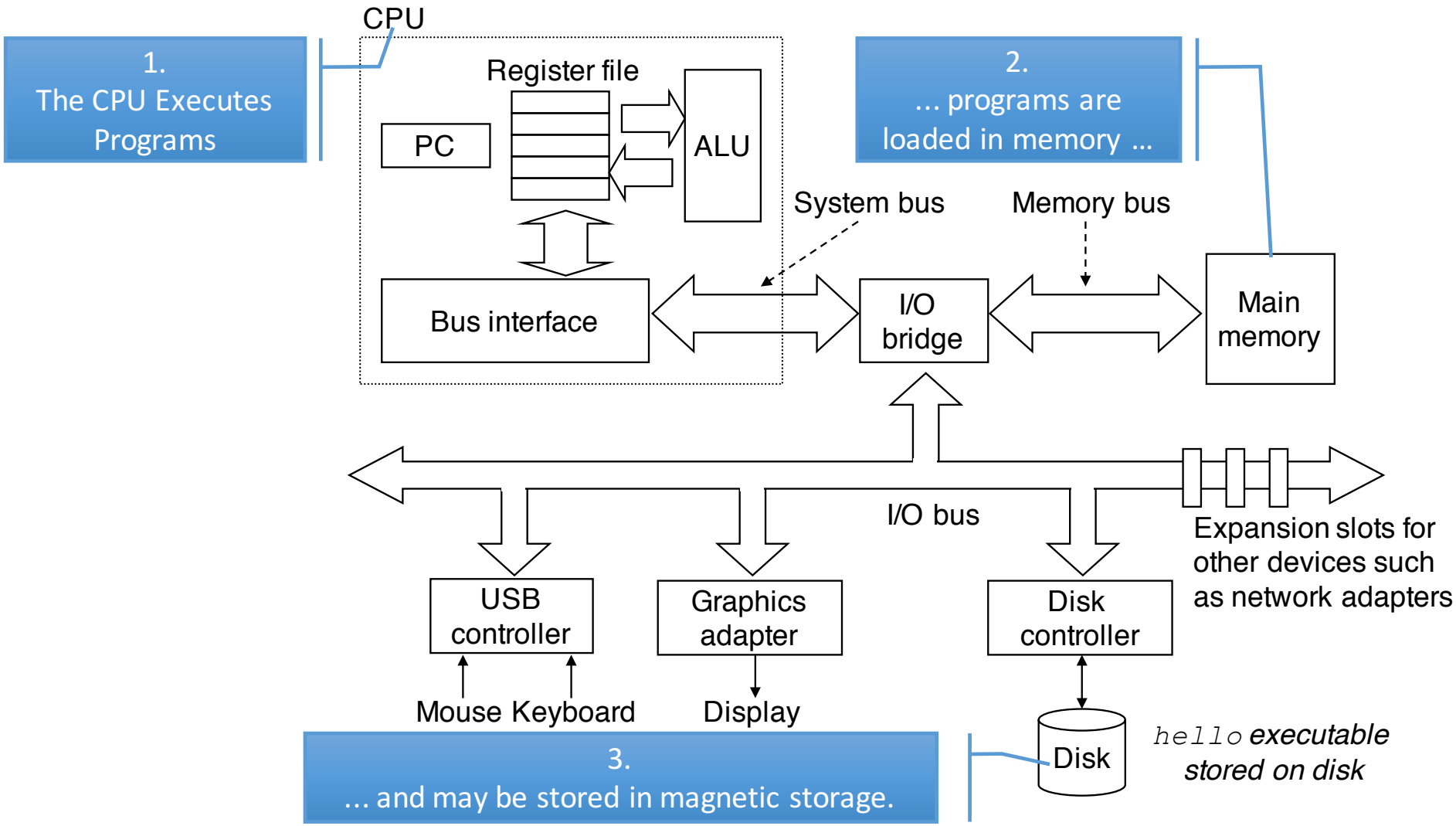
Luís Nogueira

Departamento de Engenharia Informática
Instituto Superior de Engenharia do Porto

lmn@isep.ipp.pt

2025/2026

Typical system organization



Computing systems and programs

- Computers execute *machine code*, sequences of bytes encoding the low-level operations that manipulate data, manage memory, read and write data on storage devices, and communicate over networks
- When programming in a high-level language such as C, and even more so in Java, we are shielded from the detailed machine-level implementation of our program
- In contrast, when writing programs in *assembly code*, a textual representation of the machine code, a programmer must specify the low-level instructions the program uses to carry out a computation
  - Are usually very simple since they are implemented in hardware and must be executed fast
- Therefore, a program written in a high-level language can be compiled and executed on a number of different machines, whereas assembly code is highly machine specific

RISC-V

- In the last few years, we've seen an explosion of RISC-V CPU designs on FPGA (Field-Programmable Gate Arrays) and ASIC (Application-Specific Integrated Circuits) platforms
  - RISC-V is expected to pose real competition to x86 and ARM architectures to become at least one of the dominant players in the sector
- RISC-V handles processors of different sizes with separate but consistent instruction sets:
  - RV32 - 32-bit RISC-V with 32 general-purpose registers
  - RV64 - 64-bit RISC-V with 32 general-purpose registers
  - RV32E - Reduced 32-bit RISC-V with 16 general-purpose registers for embedded systems
- We'll be focusing on the 32-bit version, but the other instruction sets work in a similar way
  - The base integer instruction set for RV32 is **RV32I** ("I"stands for integer)
  - RV32I contains the essential RISC-V instructions, including arithmetic, memory access, and flow control (**only has 40 instructions!**)
  - **RV32IM** extends RV32I with integer multiply and divide instructions, providing hardware support for efficient 'MUL', 'DIV', and remainder operations

Assembly example

Listing 1: Simple Assembly Example

```
.section .data              # section identifier: initialized data

.global myint
myint:                      # variable identifier (myint)
  .word 5                   # 4 bytes, initialized to 5

.section .text              # section identifier: code

.global func
func:                       # function definition (func)

    la   t0, myint          # load address of variable to register
    lw   t1, 0(t0)          # load value of variable (in memory) to register
    addi t1, t1, 1          # add 1 to register
    sw   t1, 0(t0)          # copy value from register to variable (in memory)

    ret
```

Why study machine-level programming?

- Clarify and help understand how:
  - High-level language code gets translated into machine language;
  - A program interfaces with the hardware (processor, memory, external devices) and operating system;
  - Data is represented and stored in memory and on external devices;
  - The processor accesses and executes instructions and how instructions access and process data.
- An important skill for serious programmers:
  - To recode in assembly language sections that are performance-critical;
  - To debug/understand the behaviour of programs for which no source code is available (for example, malware).

Assembly

- Based in instructions, registers, memory addresses, and labels
  - **Instructions** recognized by the processor
  - **Registers** of the processor
  - **Memory addresses** of variables and lines of code
  - **Labels** that assume the memory address of where they are defined

- Special characters:
  - . – starts an assembler directive
  - # – starts a comment
  - : – The colon at the end is the indicator to the assembler that causes it to recognize the preceding characters as a label

Basic structure of an Assembly program

Listing 2: Basic Assemly program example

```
# Data section: initialized variables
.section .data

.equ LINUX_SYS_CALL, 0x80      # define a constant

my_string:                     # variable identifier (my_string)
    .asciz "My string"         # NUL-terminated string

# BSS section: uninitialized storage
.section .bss

   .comm buffer, 10000         # global array of 10 000 bytes
   .lcomm buffer2, 500         # local array of 500 bytes (module local)

# Text section: code
.section .text

.global sum                    # make ‘sum‘ visible to the linker
sum:                           # entry point of the function
    # ... your RV32IM instructions here ...
    ret
```

Variable declaration

- Global initialized variables declarations are made in the `.data` section
- The **data type and an initial value** must be defined
- To avoid memory alignment issues, bigger types (that occupy the most bytes), should be declared first, then declare other variable types that occupy less, and then define the strings (more on this in future classes)

Data types

.dword – double word, 64 bits (8 bytes) integer

.word – word, 32 bits (4 bytes) integer

.half – half word, 16 bits (2 bytes) integer

.byte – 8 bits (1 byte) integer

.ascii – string (with no automatic trailing zero byte)

.asciz – string automatically terminated by zero (The "z" stands for "zero")

.float – floating point number (4 bytes)

.double – floating point number with double precision (8 bytes)

Important note

Being familiar with these names is a great help in understanding loads and stores when accessing memory later in the semester.

Tip: A word is always 32 bits wide, even on 64-bit RISC-V (RV64)

Variable declaration examples (1/2)

- Declaring an integer (4 bytes) using the `.word` directive

```
number:                 # variable name (a label)
    .word 5             # initialization value
```

- Declaring a string with the `.asciz` directive

```
message:                            # variable name (a label)
    .asciz "Hello, World!"          # initialization value
```

Variable declaration examples (2/2)

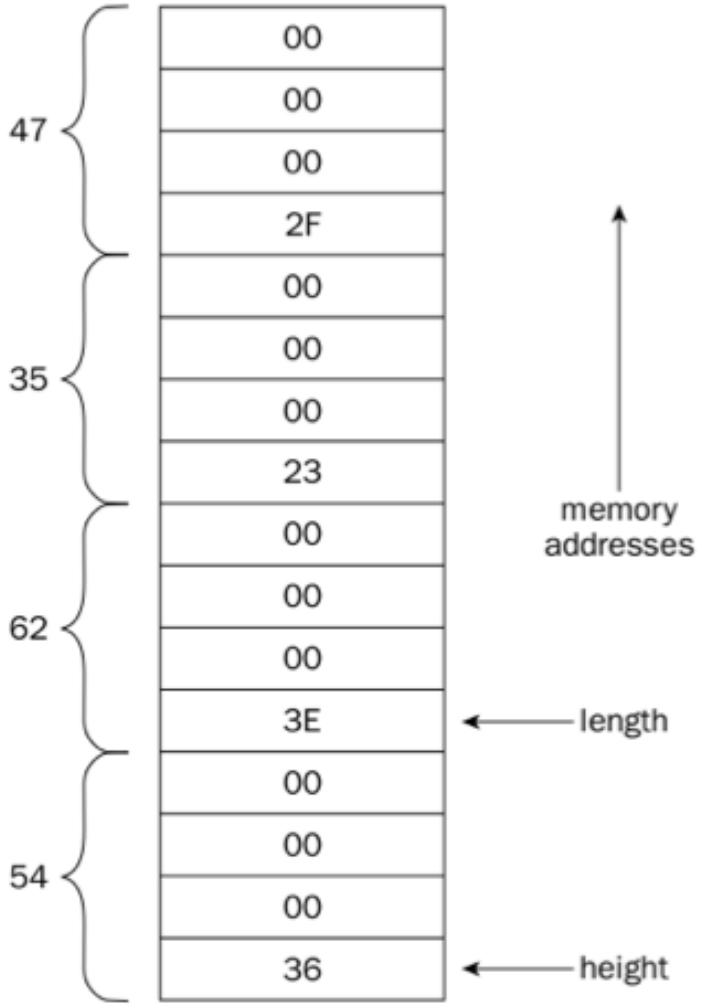
```
.section .data

factors:
    .double 37.45, 45.33, 12.30

height:
    .word 54

length:
    .word 62, 35, 47

msg:
    .asciz "This is a test message"
```



Defining constants

- The `.data` section can also be used to define constants
- Unlike variables, defining a constant does not result in reserving memory space in the final program
- Constants are replaced by their value during the generation of the code. They make code easier to read and to maintain
- Declaration example:

```
    .equ FACTOR, 3
    .equ LINUX_SYS_CALL, 0x80
```

- Usage example:

```
    li t0, LINUX_SYS_CALL
```

Reserving generic memory areas

- The `.bss` (*Block Started by Symbol*) was initially a keyword to allocate a block of *uninitialized* data of arbitrary size
- Now, most operating systems initialize data in the `.bss` segment to zero

Directive	Description
`.comm`	Declares a global memory area
`.lcomm`	Declares a local memory area

- `.comm` reserves space in the BSS segment and makes the symbol visible to the linker
  - You can refer to it across multiple object files
- `.lcomm` also reserves space in the BSS and makes the symbol visible to the linker
  - It's only visible within the current assembly file

Reserving generic memory areas: Example

```
.section .bss
  .lcomm buffer, 10000
```

- The above example declares a memory area of 10000 bytes with the identifier `buffer`
- The identifier `buffer` can only be referenced by code belonging to the same module, as it was declared with `.lcomm`

RISC-V registers

- Registers are places **within the CPU** where data is stored and is effectively available immediately
- RV32 has **32 general-purpose registers**: `x0` to `x31`
  - These registers are 32-bit wide
  - They form an integral part of the CPU design and accessing them is easy and fast. That means much faster than regular memory
- `x0` is hard-wired to 0 (zero). **Values written to it are discarded**
- You can use the other registers as you see fit, but there is an ABI (Application Binary Interface) to make life easier for programmers and allow code from different developers to interoperate
  - We will cover all the ABI registers when we detail functions later in the semester

The power of x0

- RISC-V dedicates the zero (x0) register to the number zero
- At first glance, this appears wasteful, but zero is used in many places and having it permanently available simplifies the instruction set
- Other architectures, such as MIPS and ARM64, have a zero register, and mainframe computers, such as the CDC 6600 and IBM System/360 used a zero register in the 1960s!
- As we'll see, many pseudoinstructions use the zero register, but you'll find the zero register used across RISC-V

RISC-V ABI

Register	ABI Name	Description	Saver
x0	**zero**	**Zero constant**	—
x1	ra	Return address	Callee
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	—
x4	tp	Thread pointer	—
x5-x7	**t0-t2**	**Temporaries**	**Caller**
x8	s0 / fp	Saved / frame pointer	Callee
x9	s1	Saved register	Callee
x10-x11	**a0-a1**	**Function arguments/return values**	**Caller**
x12-x17	**a2-a7**	**Function arguments**	**Caller**
x18-x27	s2-s11	Saved registers	Callee
x28-x31	**t3-t6**	**Temporaries**	**Caller**

RISC-V registers

- Assembly instructions typically use the **special name** - e.g., `t1` - for clarity, but they may also use the register number (e.g., `x6` for register number 6)

Important note

Examples in the first classes only use `x0` and the ABI **temporary registers** `t0-t6` **and the function arguments** `a0-a7` (the caller save registers)

- Until we cover how to use the stack, **do not use** any callee-saved registers, and also do not overwrite `x3` (gp) or `x4` (tp)
- This is particularly important if you call your functions from other programmer's Assembly or C code (e.g., unit tests)

RV32 Instruction Set: Key characteristics

- **Fixed-length, 32-bit instructions**
  - Every instruction occupies exactly 32 bits (4 bytes)
  - Simplifies instruction fetch and decode logic
- **Load/Store architecture**
  - Only load and store instructions access memory directly
  - All arithmetic/logical ops use registers as operands
- **RISC-style instruction formats**
  - Six base formats: R-type (register-register), I-type (immediate), S-type (store), B-type (branch), U-type (upper immediate), J-type (jump)
  - 12-bit immediate fields in I/S/B formats (signed)
- **Little-endian, aligned access**
  - Multi-byte values stored least significant byte first
  - Words must be 4-byte aligned; half-words 2-byte aligned

The `LI` Instruction

- The `LI` (load immediate) instruction is used as a way to load an immediate (constant) value into a register
- Usage: `li rd, imm`
- Where *rd* is the destination register and *imm* is a 32-bit immediate value

Pseudoinstructions

- RISC-V registers are 32 bits wide and RISC-V instructions are 32 bits wide
- An instruction needs room for an opcode and registers, so it can't hold a 32-bit immediate!
- Load immediate is not a RISC-V instruction but a **pseudoinstruction**
- Pseudoinstructions are translated into one or more real instructions by the assembler
  - Pseudoinstructions are syntactic sugar that makes code easier to write and understand, and we'll see examples of them throughout the semester
- However, before we can properly understand the details of load immediate, we need to cover arithmetic in the next class

LI instruction examples

Listing 3: Assignment of a constant (immediate) value to a register

```
# li (load immediate) is a pseudoinstruction:
# with small values (-2048 ... +2047) is replaced by a single addi
# with larger values is replaced by lui + addi

li   t0,  2             # decimal 2 (addi  t0, x0, 2)
li   t1,  42            # decimal 42 (addi t1, x0, 42)
li   t2,  -1            # -1 sign-extended to 0xFFFFFFFF (addi t2, x0, -1)

li   t3,  0x100000      # expands to:
                        #    lui  t3, 0x100       (t3 = 0x10000000)
                        #    addi t3, t3, 0

li   t4,  4100          # fits in 12-bit signed? no, so use lui+addi similarly


li   t5,  0b10101101    # binary literal (0xAD = 173 decimal)
```

The `MV` pseudoinstruction

- The `MV` (move) copies the contents of source register `rs` into destination register `rd`
- Usage: `mv rd, rs`
- Where *rd* is the destination register and *rs* is the source register

MV instruction examples

Listing 4: Copy the content of one register to another register

```
mv    t0, t1          # copy t1 into t0 (temporary computation)
mv    a0, t1          # return: move result from t1 into a0

mv    t0, x0          # clear t0 (set to 0)
mv    a0, x0          # return 0

li    t5, 0x12345     # load a 20-bit constant into t5
mv    a0, t5          # return that constant in a0

mv    t6, t5          # copy t5 into t6 (further computation)
mv    a0, t6          # return final value from t6
```

Building programs with Assembly and C functions

- Often time-critical routines are written in Assembly, and the rest of the software is written in C, thus a "mixed Assembly and C" project must be created
- When you intend to mix Assembly and C source files in a single application, the following issues are important:
  - Accessing Assembly variables in a C source file
  - Accessing C variables in an Assembly source file
  - Invoking an Assembly function in an C source file
  - Invoking a function developed in C or Assembly in an Assembly source file
  - Parameter passing scheme
  - Return value

We will (for now) write functions in Assembly that:

- Are invoked in C
- Receive up to eight integer parameters
- Do not access any global or local variables (either declared in C or Assembly)
- Do not invoke any other function

The RV32 Application Binary Interface

- The RV32 Application Binary Interface (ABI) describes the conventions for RV32 code running on Linux systems
- This includes rules about how function arguments are placed, where return values go, what registers functions may use, how they may allocate local variables, and so forth
- Calling conventions constrain both *callers* and *callees*. A caller is a function that calls another function; a callee is a function that was called (the currently-executing function is a callee)

Note

We will discuss several details during the semester, but for now you only have to consider how GCC passes parameters to a function, how to return a value, and which registers can be freely used within a function

Passing arguments

- The RISC-V calling convention passes arguments in registers when possible
- **Up to eight integer registers,** `a0-a7` **are used for this purpose**
- Arguments that are at most 32 bits wide are passed in a single argument register
  - Arguments narrower than 32 bits are widened according to the sign of their type up to 32 bits
- Arguments that are 64 bits wide are passed in a pair of argument registers
  - With the low-order 32 bits in the lower-numbered register and the high-order 32 bits in the higher-numbered register
- If you end up needing more than eight 32-bit slots, the remaining arguments go on the stack (more on this in future classes)

Return value

- A function that returns any value up to 32 bits in size places its result in register `a0`
- A 64-bit result is split across registers `a0` (low 32 bits) and `a1` (high 32 bits)

Example: Putting it all together - C source

Listing 5: main.c

```c
#include <stdio.h>
#include "asm.h"

int main(void) {
    int res, op1  = 0xDEADC0DE;
    long long op2 = 0xFEEDFACECAFEBEEF;
    short op3     = 0xBEEF;

    res = func(op1, op2, op3);

    printf("res = %x\n", res);

    return 0;
}
```

Listing 6: asm.h

```c
#ifndef ASM_H
#define ASM_H
int func(int, long long, short);
#endif
```

Example: Putting it all together - Assembly source

Listing 7: asm.s

```
.section .text
.global func            # define func as global

# int func(int op1, long long op2, short op3)
func:
    # op1 in a0
    # op2 in a1 (low 32 bits) and a2 (high 32 bits)
    # op3 in a3 (sign-extended to 32 bits)


    # body of the function
    ...

    # return value in a0
    li a0, 42

    # return from the function
    ret
```

Example: Putting it all together - Makefile

Listing 8: Makefile

```
CC=riscv32-linux-gcc
CFLAGS=-Wall -Wextra -fanalyzer -g
ARCHFLAGS=-march=rv32imd -mabi=ilp32d
LIBRV32=${HOME}/bin/bootlin/riscv32-buildroot-linux-gnu/sysroot

program.elf: main.o asm.o
  $(CC) $(CFLAGS) $(ARCHFLAGS) main.o asm.o -o program.elf

main.o: main.c
  $(CC) $(CFLAGS) $(ARCHFLAGS) -c main.c

asm.o: asm.s asm.h
  $(CC) $(CFLAGS) $(ARCHFLAGS) -c asm.s

clean:
  rm -f *.o *.elf

run: program.elf
  qemu-riscv32 -L $(LIBRV32) ./program.elf
```