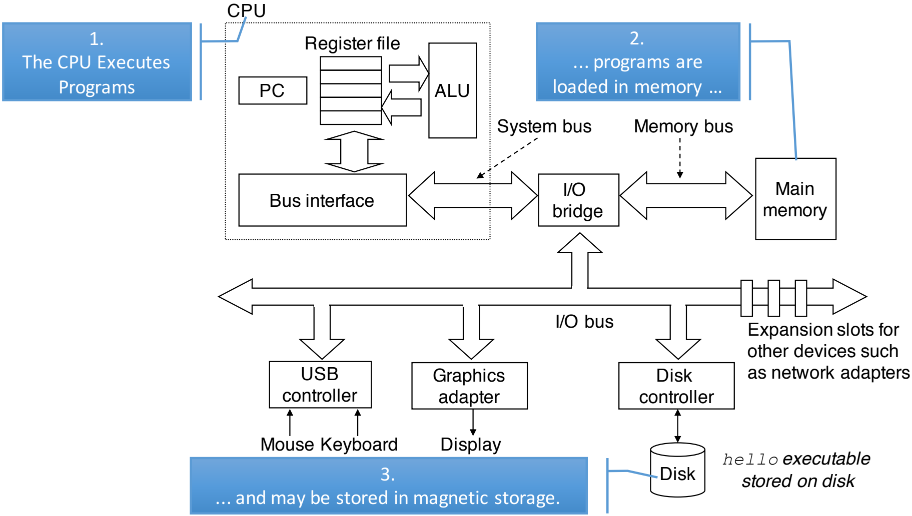
Computer Architecture (Practical Class)
Introduction to Assembly

Luís Nogueira

Departamento de Engenharia Informática
Instituto Superior de Engenharia do Porto

lmn@isep.ipp.pt

2023/2024


1.
The CPU Executes
Programs
CPU
Register file
PC
ALU
Bus interface
System bus
Memory bus
I/O
bridge
Main
memory
2.
... programs are
loaded in memory ...
I/O bus
Expansion slots for
other devices such
as network adapters
USB
controller
Graphics
adapter
Disk
controller
Mouse Keyboard
Display
Disk
hello executable
stored on disk
3.
... and may be stored in magnetic storage.

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

Listing 1: Simple Assembly Example

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

myint:                    # variable identifier (myint)
    .int 5                # integer, initialized to 5

.section .text            # section identifier: code

func:                     # function definition (func)

    movl myint(%rip), %eax  # copy variable (in memory) to register
    addl $1, %eax           # add 1 to register
    movl %eax, myint(%rip)  # 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).

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

- Special characters:

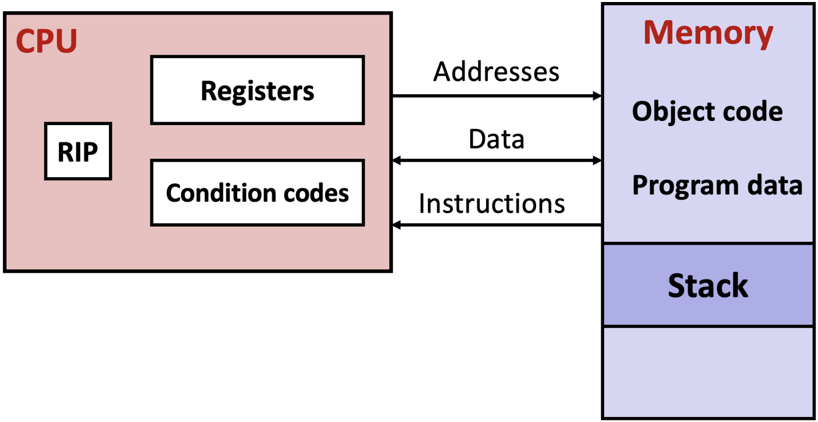
  . – starts an assembler directive
  # – starts a comment
  % – starts a register name
  $ – starts an immediate value

Basic structure of an Assembly program

Listing 2: Basic Assemly program example

```
# the data section allows to declare initialized variables
.section .data # the ".section" can be ommitted

        .equ LINUX_SYS_CALL, 0x80       # the .equ directive defines a
                                        # constant
output_int:
        .asciz "My string"      #definition of a string

# the bss section is used to define uninitialized memory areas
.section .bss

        .comm buffer, 10000     # global array of 10000 bytes
        .lcomm buffer2, 500     # array of 500 bytes, only visible in
                                # current module (source file)

# the text section has the assembly instructions
.section .text

        .global sum             #defines the function as global

sum:        # start of the function
    ...     # instructions
    ret
```

Assembly's programmer view



Variable declaration

- Global 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), should be declared first, then declare other variable types that occuppy less, and then define the strings (more on this in future classes)

Data types

- .octa – 128 bits (16 bytes) integer
- .quad – 64 bits (8 bytes) integer
- .long – the same as .int
- .int – 32 bits (4 bytes) integer
- .short – 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

Notice the difference between the long type in C and in Assembly. Use the type .quad for an 8 byte integer in Assembly

Variable declaration examples (1/2)

- Declaring an integer using the `.int` directive

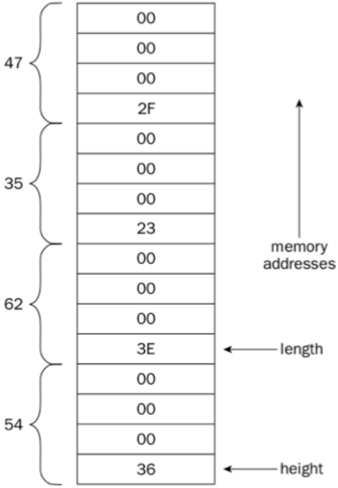
```
number:                 # variable name
        .int 5          # initialization value
```

- Declaring a string with the `.asciz` directive

```
message:                                # variable name
    .asciz ‘‘Hello, World!’’            # initialization value
```

Variable declaration examples (2/2)

```
.section .data

factors:
    .double 37.45, 45.33, 12.30

height:
    .int 54

length:
    .int 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:

```
    movq $LINUX_SYS_CALL, %rax
```

Reserving generic memory areas

- The `.bss` (*Block Started by Symbol*) can be used to reserve *uninitialized* memory areas of arbitrary size

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

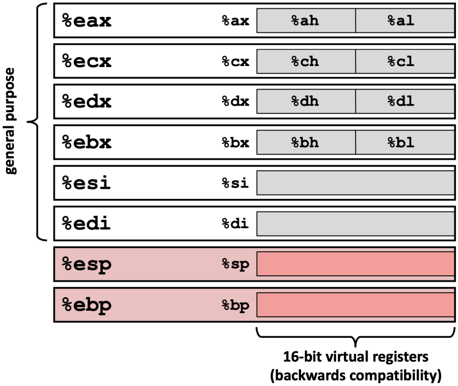
- Example Declaration

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

- The above 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`

IA32 registers

- There are eight 32-bit registers **EAX, EBX, ECX, EDX, EBP, ESP, EDI** and **ESI** for temporary usage
- The 16-bit registers **AX, BX, CX, DX, BP, SP, DI** and **SI** are contained in the corresponding 32-bit registers and represent their 16 less significant bits
- The 8-bit registers **AH, BH, CH, DH** are contained in the corresponding 16-bit registers (AX, BX, CX, DX) and represent their 8 most significant bits
- The 8-bit registers **AL, BL, CL, DL** are contained in the corresponding 16-bit registers (AX, BX, CX, DX) and represent their 8 less significant bits



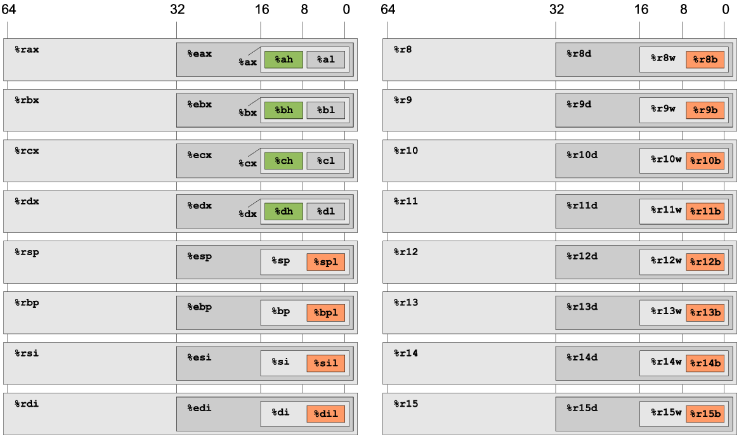
x86-64 registers

- In the extension to x86-64, the original eight registers were expanded to 64 bits, labeled *%rax* through *%rbp*
- In addition, eight new registers were added, and these were given labels according to a new naming convention: *%r8* through *%r15*
- Portions of all 16 registers can be accessed as byte (8-bit), word (16-bit), double word (32-bit), and quad word (64-bit) quantities

Important notes

- A set of standard programming conventions governs how the registers are to be used for managing the stack, passing function arguments, returning values from functions, and storing local and temporary data
- We will cover all these topics throughout the semester

x86-64 registers



Operand specifiers

- Most instructions have one or more operands specifying the source values to use in performing an operation and the destination location into which to place the result
- The x86-64 registers, memory operations and instructions use the following data types (among others):

Data type	**Suffix**	Size (bytes)
byte	b	1
word	w	2
long (double word)	l	4
quad word	q	8

The `MOV` Instruction

- The `MOV` instruction is used as a way to copy data
- Usage: `mov origin, destination`
- *origin* can be a memory address, a constant (immediate) value or a register
- *destination* can be a memory address or a register
- The size of the data to be copied **must be indicated** by adding a character at the end of the instruction
- The `MOV` instruction can copy values of 8(b), 16(w), 32(l), or 64 (q) bits

Important notes

- Two memory addresses cannot be used simultaneously
- Origin and destination must be of the same size

The `MOV` Instruction - Effects on destination

- For most cases, the mov instructions will only update the specific register bytes or memory locations indicated by the destination operand
- The only exception is that when `movl` has a register as the destination, it will also set the high-order 4 bytes of the register to 0
- This exception arises from the convention, adopted in x86-64, that **any instruction that generates a 32-bit value for a register also sets the high-order portion of the register to 0**

MOV instruction examples (1/2)

Listing 3: Assignment of constant (immediate) value

```
movq $5236, %rax    # moves the integer value 5236 to RAX

movl $-345, %ecx    # moves the integer value -345 to ECX
                    # the 32 most significant bytes of RCX are set to zero

movw $0xFFB1, %dx   # moves the value -79 (0xFFB1 in hexadecimal)
                    # to the least significant 16 bits of RDX

movb $0x0A, %al     # moves the value 10 (0x0A in hexadecimal)
                    # to the least significant byte of RAX
```

- x86-64 code often refers to global variables using **%rip-relative addressing**: a global variable named *a* is referenced as *a(%rip)* rather than *a*
- This style of reference supports *position-independent code*, a security feature. It specifically supports position-independent executables (PIE), which are programs that work independently of where their code is loaded into memory
- In a PIE, the operating system loads the program at varying locations: every time it runs, the program's functions and global variables have different addresses. This makes the program harder to attack (though not impossible)
- Therefore, global variables are referenced relatively to the current value of the program counter (the %rip register in x86-64)
- We will dive into the details of memory addressing in the next classes

MOV instruction examples (2/2)

Listing 4: Copying the contents of a variable to a register and vice-versa

```
.section .data

#declare a variable called 'myinteger'
myinteger:
    .int 5

.section .text

function:
  movl myinteger(%rip), %eax  # copy value of variable (in memory) to register
  ...                         # do something with the value...
  movl %eax, myinteger(%rip)  # copy register value to variable (in memory)
  ...
  ret
```

The MOVABSQ Instruction

- The regular `movq` instruction can only have immediate source operands that can be represented as 32-bit two's-complement numbers
- This value is then sign extended to produce the 64-bit value for the destination
- The `MOVABSQ` (move absolute quad word) instruction can have an arbitrary 64-bit immediate value as its source operand and can only have a register as a destination
- Usage: `movabsq imm, register`

- As described, there are two different conventions regarding whether and how data movement instructions modify the upper bytes of a destination register
- This distinction is illustrated by the following code sequence:

Listing 5: Understanding how data movement changes a destination register

```
movabsq $0x0011223344556677, %rax # %rax = 0x0011223344556677
movb $-1, %al                     # %rax = 0x00112233445566FF
movw $-1, %ax                     # %rax = 0x001122334455FFFF
movl $-1, %eax                    # %rax = 0x00000000FFFFFFFF
movq $-1, %rax                    # %rax = 0xFFFFFFFFFFFFFFFF
```

The `MOVZ` class of instructions

- The `movz` class fills out the remaining bytes of the destination with zeros
- Each instruction name has size designators as its final two characters—the first specifying the source size, and the second specifying the destination size
- Usage: `movz[bw|bl|bq|wl|wq] source,destination`

Instruction	**Description**
movzbw	Move zero-extended byte to word
movzbl	Move zero-extended byte to double word
movzbq	Move zero-extended byte to quad word
movzwl	Move zero-extended word to double word
movzwq	Move zero-extended word to quad word

Important notes

- Note the absence of an explicit instruction to zero-extend a 4-byte source value to an 8-byte destination
- This type of data movement can be implemented using a `movl` instruction having a register as the destination

The `MOVS` class of instructions

- Instructions in the `movs` class fill out the remaining bytes of the destination by sign extension, replicating copies of the most significant bit of the source operand
- Each instruction name has size designators as its final two characters—the first specifying the source size, and the second specifying the destination size
- Usage: `movs[bw|bl|bq|wl|wq|lq] source,destination`

Instruction	Description
movsbw	Move sign-extended byte to word
movsbl	Move sign-extended byte to double word
movsbq	Move sign-extended byte to quad word
movswl	Move sign-extended word to double word
movswq	Move sign-extended word to quad word
movslq	Move sign-extended double word to quad word

Example: Comparing byte movement instructions

- The following example illustrates how different data movement instructions either do or do not change the high-order bytes of the destination
- Observe that the three byte-movement instructions `movb`, `movsbq`, and `movzbq` differ from each other in subtle ways

Listing 6: Comparing byte movement instructions

```
movabsq $0x0011223344556677, %rax # %rax = 0x0011223344556677
movb    $0xAA, %dl                # %dl  = 0xAA
movb    %dl, %al                  # %rax = 0x00112233445566AA
movsbq  %dl, %rax                 # %rax = 0xFFFFFFFFFFFFFFAA
movzbq  %dl, %rax                 # %rax = 0x00000000000000AA
```

Practice

- Read the document “Building programs with Assembly and C functions” available in Moodle