Computer Architecture (Practical Class)
Introduction to Assembly

Luís Nogueira Raquel Faria

Departamento de Engenharia Informática
Instituto Superior de Engenharia do Porto

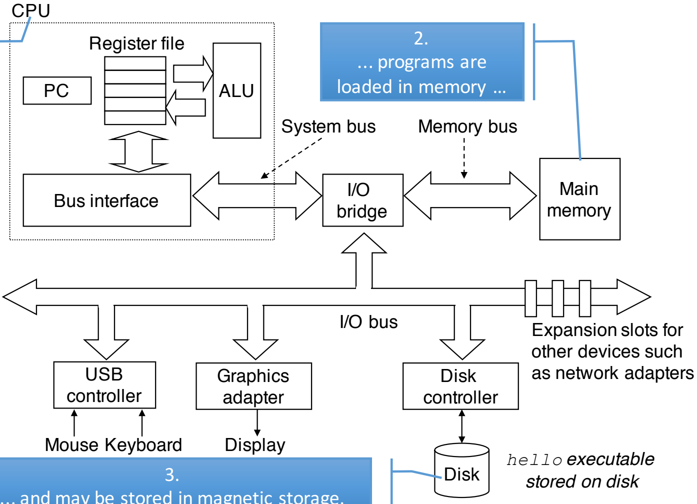
{lmn,arf}@isep.ipp.pt

2020/2021

Typical system organization

1.
The CPU Executes Programs

2.
… programs are loaded in memory …



3.
… and may be stored in magnetic storage.

- Computing systems are composed of hardware and system software (the Operating System) that work together to execute programs (user applications)
- A program can be seen as a cooking recipe. It has a list of ingredients (the *variables*) and a list of steps (the *instructions*) that tell the computer what to do with the variables. Variables can represent numbers, text, sounds, images, etc.

Program = Data + Instructions

- A combination of computer instructions and data allow the hardware to execute computing tasks

The CPU is a Digital Machine (1/3)

- Q: What type of data can be used?
- A: Binary data

Modern CPUs process binary data

- In order to be manipulated, all data, such as instructions, numbers, symbols, pictures, etc, must be represented as binary digits
- A binary digit is called a bit and represents either a 0 or a 1

Note: A previous course (Princípios da Computação) detailed several numeric representation systems (such as binary, hexadecimal or octal). Please review these materials as needed

- Q: What instructions are supported by the CPU?
- A: Depends on the particular CPU. It is necessary to read the CPU manual!

Machine code

- The CPU instructions are called *machine code* because they are specific to a CPU type or model
- Are usually very simple since they are implemented in hardware and must be executed fast
- Machine code instructions are also represented as binary data!

- They usually include instructions to perform:
  - logic and arithmetic;
  - read/write from I/O devices;
  - flow control;
  - moving data around.

- C code example

Program = Data + Instructions

```
int sum ( int a, int b ) {
      return a + b;
}
```

- The same example in machine code for an Intel CPU

Program = Binary Data + Machine Code Instructions

```
55 89 e5 8b 0c 03 45 08 89 ec 5d c3
```

This is very hard to read! It is *Machine Code*; only the machine needs to understand it...

Machine-level coding

One can think of developing at machine-level in several ways; For example:

- Read the CPU manual and toggle switches to positions corresponding to the desired microprocessor instruction operation code (opcode) in binary.
  - Example: Altair 8800 (1975)
- Read the CPU manual and used a keyboard to enter the desired microprocessor instruction opcodes in hexadecimal.
  - Example: Kim-1 (1976)
- Define a name (mnemonic) for each instruction, write the instructions in a file, and use a program that generates the corresponding machine code.
  - Program: *Assembler*
  - Language: *Assembly*

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

funcao:                     # function definition (funcao)

    movl myint, %eax        # copy variable to register
    addl $1, %eax           # add 1 to register
    movl %eax, myint        # copy value from register to variable

    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.
- While most new software is developed in high-level languages, which are easier to write and maintain, assembly can be used, for example:
  - 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 and labels
  - **Instructions** recognized by the processor
  - **Registers** of the processor
  - **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 a value

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 "imprimir inteiro:"      #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
```

Variable declaration

- Variable declarations can be made in the `.data` section
- The data type must be defined
- To avoid memory alignment issues, integer types (that occupy the most), should be declared first, then declare other variable types that occuppy less, and then defined the strings

Data types

- .octa – 128 bits (16 bytes) integer
- .quad – 64 bits (8 bytes) integer
- .double – floating point number with double precision (8 bytes)
- .long – 32 bits (4 bytes) integer
- .int – 32 bits (4 bytes) integer
- .float – floating point number (4 bytes)
- .short – 16 bits (2 bytes) integer
- .byte – 8 bits
- .ascii – string
- .asciz – string automatically terminated by zero

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 ‘‘mensagem de teste’’        # initialization value
```

Variable declaration examples (2/2)

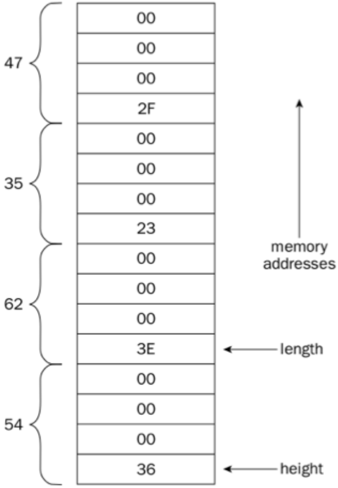
```
.section .data

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
  - Constants are replaced by their value during the generation of the code. They make code easier to read and to maintain
- Note that defining a constant does not result in reserving memory space in the final program.
- Declaration example:

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

- Usage example:

```
movl $LINUX_SYS_CALL, %eax
```

Reserving generic memory areas

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

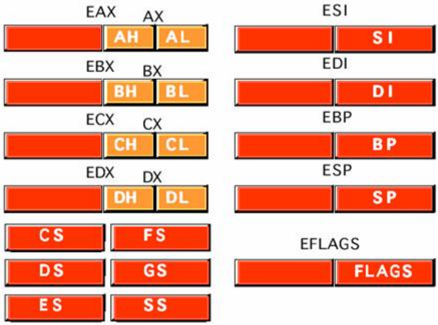
Directive	Description
`.comm`	Declares a global memory area
`.lcomm`	Declares a local memory area

- 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`

- The 32-bit registers **EAX, EBX, ECX, EDX, EBP, ESP, EDI** and **ESI** are generic registers 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



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 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 numbers of 8(b), 16(w) or 32(l) bits

Important notes

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

Listing 3: Assignment of constant value

```
movl $-345, %ecx    # moves the integer value -345 to ECX

movw $0xffb1, %dx   # moves the value -79 (0xffb1 in decimal)
                    # to the least significant 16 bits of EDX

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

MOV instruction examples (2/2)

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

```
...

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

...

    movl myinteger, %eax   # copy value of variable to register
    ...                    # do something with the value...
    movl %eax, myinteger   # copy register value to variable

...
```

Important note about the $ sign

- Using the $ in the text section requires some care

`122` – the content of memory address 122
`$122` – the number 122

`variable` – the contents pointed by memory address in `variable`
`$variable` – the memory address of the label

Arithmetic operations: ADD

- The `ADD` instruction adds two integers
- Usage: `add` *origin*, *destination*
- Performs the operation *destination* = *destination* + *origin* (the result is placed in *destination*
- *origin* can be a memory address, a constant value or a register
- *destination* can be a memory address or a register
- a memory address for *origin* and *destination* cannot be used simultaneously
- the `ADD` instruction can add numbers of 8(b), 16(w) or 32(l) bits

ADD instruction examples

Listing 5: Adding bytes, words, and longs

```
addb $10, %al      # adds 10 to the 8-bit AL register; AL=AL+10
addw %bx, %cx      # adds the value of BX to CX (16 bits); CX=CX+BX
addl var1, %eax    # adds the 32-bit value in var1 to EAX; EAX=EAX+var1
addl %eax, %eax    # adds EAX to itself; EAX=EAX+EAX
```

Arithmetic operations: SUB

- The `SUB` instruction subtracts two integers
- Usage: `sub` *origin*, *destination*
- performs the operation *destination* = *destination* - *origin* (the result is placed in *destination*
- *origin* can be a memory address, a constant value or a register
- *destination* can be a memory address or a register
- a memory address for *origin* and *destination* cannot be used simultaneously
- the `SUB` instruction can add numbers of 8(b), 16(w) or 32(l) bits

SUB instruction examples

Listing 6: Subtracting bytes, words, and longs

```
subl $10, %eax     # subtract 10 to the current value of EAX; EAX=EAX-10
subw %bx, %cx      # subtract the value of BX to CX (16 bits); CX=CX-BX
subl var1, %eax    # subtract the 32-bit value in var1 to EAX; EAX=EAX-var1
subl %ecx, %eax    # subtract ECX to EAX; EAX=EAX-ECX
```

Arithmetic Operations: INC and DEC

- The `INC` and `DEC` instructions increment (INC) and decrement (DEC) an integer by one, respectively
- Usage: `inc` *destination*
- performs the operation $destination = destination + 1$;
- *destination* can be a memory address or a register
- the `INC` and `DEC` instructions can be used in numbers of 8(b), 16(w) or 32(l) bits

INC/DEC instructions examples

Listing 7: Increment/Decrement bytes, words, and longs

```
incl %eax    # EAX=EAX+1 (32 bits)

incw %bx     # BX=BX+1 (16 bits)

decb %cl     # CL=CL-1 (8 bits)
```

Sharing variables between Assembly and C (1/3)

- We will (for now) write functions in Assembly that receive no parameters
- Then, we write programs in C that call our Assembly functions as if they were native C functions
- To share global variables between C and Assembly we will use the `extern` C keyword
  - It declares to the compiler that a variable is defined (the memory is reserved) in another source file (in our case, in the Assembly source file(s))
- To make our Assembly functions return:
  - a 32-bit value, leave that return value in the `%eax` register
  - a 64-bit value, leave the return value in the `%edx:%eax` registers

Sharing variables between Assembly and C (2/3)

The `extern` C Keyword can be used in different ways to share variables between C and Assembly. The following is a recommended practice, that avoids common problems

- On the C source, we:
  1. Declare the *functions and variables* implemented in Assembly and used in C in a separate `.h` file (often called `asm.h`). Declare Assembly variables using the `extern` C keyword (functions are `extern` by default):

     Listing 8: asm.h

     ```
     int asm_function();
     extern int asm_integer;
     ```

  2. Use the keyword `#include` to include the previous `.h` file in the C source files (`.c` files) that use the Assembly functions or variables, and use the Assembly functions/variables like native C functions/variables:

     Listing 9: main.c

     ```
     #include "asm.h"
     ...
     int main() {
       ...
       asm_integer=10;
       asm_function();
       ...
     }
     ```

- On the Assembly source, we:
  1. Declare the variables and functions used by the C sources and define them as visible using the `.global` directive, and
  2. Leave the return value on the %eax register to return a 32-bit value, or in the %edx:%eax registers for a 64-bit value.

Listing 10: asm.s

```
.section .data
asm_integer:            # variable declaration
    .int 5
.global asm_integer     # define variable as global

.section .text
.global asm_function    # define function as global
asm_function:           # start of the function
    ...
    movl $0, %eax       # reaching here, will return 0
                        # (eax will not be changed until ret)
    ...

    ret
```

Prologue and Epilogue

Important note on writing Assembly functions

- When a function executes, it needs to perform some setup/cleanup code that we call prologue (at the beginning) and epilogue (at the end). Later, we will see why they are needed
- While some functions *may* behave correctly with no epilogue and prologue, **you should have a prologue and epilogue in all functions you write**.

Listing 11: Prologue/Epilogue example (`http://codepad.org/fwYPhv0n`)

```
function_label:
# prologue
    pushl %ebp          # save previous stack frame pointer
    movl %esp, %ebp     # the stack frame pointer for our function

# body of the function
    # here we implement our function...

# epilogue
    movl %ebp, %esp    # restore the stack pointer ("clear" the stack)
    popl %ebp          # restore the stack frame pointer

#return from the function
    ret
```

Example: Sum two variables - C source

Listing 12: main.c (http://codepad.org/wnUhIpuK)

```
#include <stdio.h>
#include "asm.h" // defines op1, op2 and sum_op1_op2(void)

int main(void) {
    int res=0;
    printf("Value of op1?:");
    scanf("%d",&op1);
    printf("Value of op1?:");
    scanf("%d",&op2);

    /* res = op1 + op2; */
    res = sum_op1_op2();

    printf("%d = %d + %d\n", res, op1, op2);
    return 0;
}
```

Listing 13: asm.h (http://codepad.org/lvHvUwSK)

```
int sum_op1_op2(void);
extern int op1, op2;
```

Example: Sum two variables - Assembly source

Listing 14: asm.s (http://codepad.org/0de8SKIa)

```
.section .data       # declare op1, op2
op1:
        .int 0
op2:
        .int 0
.global op1          # define op1, op2 as globals
.global op2

.section .text
.global sum_op1_op2   # define global function int sum_op1_op2(void)
sum_op1_op2:
# prologue
    pushl %ebp        # save previous stack frame pointer
    movl %esp, %ebp   # the stack frame pointer for our function

# body of the function
    movl op1, %ebx    # place op1 in ebx
    movl op2, %eax    # place op2 in eax
    addl %ebx, %eax   # add ebx to eax; the result is in eax
                      # and will be our return value
# epilogue
    movl %ebp, %esp   # restore the stack pointer ("clear" the stack)
    popl %ebp         # restore the stack frame pointer
    ret               # return from the function
```

Example: Sum two variables - Makefile

Listing 15: Makefile (`http://codepad.org/8BfSoSR8`)

```
main: main.o asm.o
	gcc -Wall -g main.o asm.o -o main

main.o: main.c asm.h
	gcc -Wall -g -c main.c

asm.o: asm.s
	gcc -Wall -g -c asm.s

run: main
	./main

clean:
	rm *.o main
```

- Note: We are always using the compiler flags `-g` and `-Wall`.

Practice

- Write a C program that calls `increment()`, a function implemented in Assembly
- Function `increment()` increments the value of the global integer variable `g_number` and returns the this value (after the increment)
- The C program should assign a test value to `g_number`, call `increment()` and then print both `g_number` and the value returned by the function
- Write a `Makefile` to compile your program