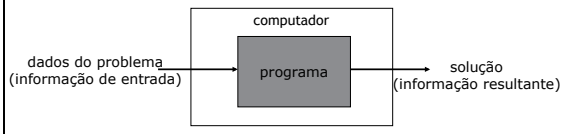
Arquitetura de Computadores

MIEI – 2017/18
DI-FCT/UNL
Aula 2

Execução de um programa

- calcula-se determinada solução (*output*) para determinado problema (*input*) usando um programa que é executado no computador



Questões:

- Como representar os dados?
- Como representar o programa?
- Que instruções?
- Como executar as instruções / programa?
- **O que é um computador?**

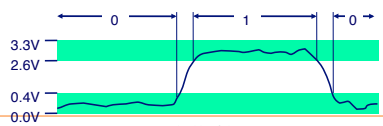
Questões (por agora):

- Como representar os dados?
- Como representar o programa?

- Tudo são bits! Porquê?
  - Porque não dígitos decimais?
  - Ou grandezas analógicas?

Codificação da informação

- A informação manipulada pelo computador tem de estar representada como este é capaz de a interpretar
- Tecnologia atual baseia-se em circuitos electrónicos digitais binários
  - os circuitos facilmente reconhecem, mantêm e transmitem dois estados: ligado/desligado
  - cada circuito num destes estados representa um bit de informação → ***bi**nary digi**t***



Codificação binária

- Tudo no computador está representado por **bits**
  - podemos convencionar que cada bit representa: ligado/desligado ou 1/0 ou verdade/falso ou …etc…
  - os circuitos podem manipular simultaneamente grupos de bits (por exemplo: 8, 12, 16, 32, etc…)→ cada arquitetura tem a **palavra** típica de trabalho
  - quem "olha para os bits" interpreta-os como quiser!
- Como representar texto, imagens, sons, etc.???
  - Definimos normas de codificação

Codificando valores lógicos

- Para os valores booleanos podemos convencionar: Verdade = 1 e Falso = 0
- Operações algébricas: conjunção/e/*and*; disjunção/ou/*or*; negação/*not*

		and
0	0	0
0	1	0
1	0	0
1	1	1

		or
0	0	0
0	1	1
1	0	1
1	1	1

	not
0	1
1	0

Operações booleanas

- O computador faz as operações boolenas sobre grupos de bits (palavras de trabalho)
- Exemplos com **nibbles** (4 bits ou *half-byte*):

```
    1 0 1 0        1 0 1 0
and 0 0 1 1    or  0 0 1 1    not 0 0 1 1
    0 0 1 0        1 0 1 1        1 1 0 0
```


Codificando números

- Notação posicional em base 10:
  - dez dígitos representados pelos símbolos: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
  - num número, cada algarismo representa uma potência de 10
  - exemplo:

$$159 = 1 \times 100 + 5 \times 10 + 9 \times 1$$

$$1x10^2 + 5x10^1 + 9x10^0$$

centenas dezenas unidades

$$\sum_{i=0}^{n-1} x_i 10^i$$

Representação binária

- Representação binária (base 2):
  - 2 dígitos ou símbolos: 0, 1
  - cada algarismo no número representa uma potência de 2
  - exemplo: 10011111

em decimal:

$$1x2^7 + 0x2^6 + 0x2^5 + 1x2^4 + 1x2^3 + 1x2^2 + 1x2^1 + 1x2^0$$

$$= 159$$

$$\sum_{i=0}^{n-1} x_i 2^i$$

Conversões

- Decompondo um número nos dígitos de determinada base
  - decompomos no polinómio em potências da base
  - exemplo:

$$159 = 1 \times 100 + 5 \times 10 + 9 \times 1$$

$$= ((1 \times 10) + 5) \times 10 + 9$$

  - ou seja, dividimos sucessivamente pela base pretendida:

```
159 |10
 9  15 |10
    5   1 |10
        1  0
```

unidades, dezenas, centenas

Conversões

- Para outras bases é idêntico…
  - binário, dividimos por 2:

```
159 |2
 1   79 |2
     1  39 |2
        1  19 |2
           1   9 |2
              1   4 |2
                  0   2 |2
                      0   1 |2
                          1   0
```

$159_{10} = 10011111_2$

Binário → bits

- Booleanos e números inteiros são facilmente representáveis no computador
  - basta converter para a representação binária e convencionar, por exemplo, 0=desligado, 1=ligado
- Podemos representar qualquer informação que consigamos converter em números → **digitalizar**
  - Convém usar normas

Vantagens e desvantagens

- Desvantagens do binário:
  - não é natural para nós;
    - as conversões, binário → base 10 → binário, não são intuitivas
  - a representação torna-se extensa.
- Mas ... os circuitos são simples e fiáveis
  - Podemos compor circuitos cada vez mais complexos à custa de blocos mais simples
  - Pode-se integrar milhões destes circuitos em pouco espaço → representação extensa não é problema

Outras bases de numeração

- Para nós pode ser mais fácil usar outras bases:
  - Representações menos extensas
  - Mas fáceis de converter para/de binário
- As candidatas são bases potências de 2
  - cada digito nessas bases corresponde diretamente a um grupo de dígitos da base 2

Representação de números (cont)

- Representação hexadecimal (base 16):
  - 16 símbolos:

    0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F
  - cada algarismo no número representa uma potência de 16
  - exemplo: 9F

    em decimal: $9 \times 16^1 + 15 = 159$
  - conversão para hexadecimal:

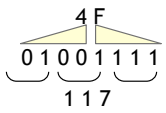
```
159 | 16
 15   9   | 16
      9     0
```


Relação entre representações

- Usando bases potências de 2 temos uma relação direta entre os dígitos de cada base
- exemplo:

  4 dig. bin. ⇔ 1 dig.hex.

  3 dig. bin. ⇔ 1 dig.octal



```
   4 F
0 1 0 0 1 1 1 1
   1 1 7
```

10	16	8	2
0	0	0	0
1	1	1	1
2	2	2	010
3	3	3	011
4	4	4	100
5	5	5	101
6	6	6	110
7	7	7	111
8	8	10	1000
9	9	11	1001
10	A	12	1010
11	B	13	1011
12	C	14	1100
13	D	15	1101
14	E	16	1110
15	F	17	1111

Operações

- As operações aritméticas são independentes das bases de representação dos números
- Exemplos em binário:

```
      1      (carry)          1  1  1   (borrow)
   1 0 1 0   (10)            1 0 1 0   (10)
+     1 1    (3)           -     1 1    (3)
   1 1 0 1   (13)              1 1 1   (7)
```

Codificação de inteiros com sinal

- Inteiros positivos e negativos em **complemento para 2**:
  - Fixa-se o número de bits
  - usa-se o bit mais significativo para codificar o sinal. O bit de sinal representa -2^n (n=nº de bits)
  - $-x$ é representado por $2^n - x$
  - E.g: em 4 bits -2 é representado como antes era o 14 (16-2 = 14)
- Propriedade importante:

  5+(−5) = 0 !

```
  0101
+ 1011
1 0000
```

Dec	c/sinal bits
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
-8	1000
-7	1001
-6	1010
-5	1011
-4	1100
-3	1101
-2	1110
-1	1111

Fazer o complemento para 2

- O complemento para 2 de um número X com n bits é dado por 2^n-X
- Também pode ser facilmente obtido por:
  - Negação dos bits, somando depois 1

    (not X) + 1
- Permite obter o complemento de qualquer número, positivo ou negativo. Exemplos:
  - Tendo 5 obter -5: 0101 → 1010 → 1011
  - Tendo -5 obter 5: 1011 → 0100 → 0101
- É sempre X+(-X) = 0 → X+(not X) +1 = -1 +1 = 0

Representação binária de inteiros (2)

- Representação binária pode ser de um número sem ou com sinal:
  - Exemplo com palavra de 5 bits:

1	0	1	1	1
4	3	2	1	0
±16	8	4	2	1

  - S/sinal: $1x2^4 + 0x2^3 + 1x2^2 + 1x2^1 + 1x2^0$ = **23**

$$\sum_{i=0}^{n-1} x_i 2^i$$

  - C/sinal: $-1x2^4 + 0x2^3 + 1x2^2 + 1x2^1 + 1x2^0$ = -16+7 = **-9**

$$-x_{n-1}2^{n-1} + \sum_{i=0}^{n-2} x_i 2^i$$

Gama de valores Inteiros

- Gama de valores representáveis em n bits:
  - 2^n valores possíveis
  - Inteiros sem sinal: de 0 a 2^n-1
  - Inteiros com sinal: de -2^{n-1} a 2^{n-1}-1
- Operações:
  - adição (normal mesmo com negativos)
  - subtração: A-B = A+(-B)
  - podem produzir *Carry* («*vai 1*») que se perde
  - se em comp. a 2 podem produzir *Overflow*

Carry e Overflow

Assumindo palavras de 4 bits

- *Carryout*: sempre que o resultado extravasa os bits disponíveis

```
  0 1 0 0   (4)
+ 1 1 0 0   (12)
1 0 0 0 0   (0?)
(carry)     (16)
```

- *Overflow* (compl. p/ 2): sempre que o resultado extravasa a representação disponível com sinal

```
  1 1 1 1   (-1)
+ 1 0 0 0   (-8)
1 0 1 1 1   (7?)
(overflow)  (-9)
```


Representação de texto

- Podemos representar as letras e outros símbolos convencionando uma codificação
  - Numeramos todos esses caracteres!
  - Temos de seguir a mesma convenção na introdução dos caracteres no computador e quando os apresentados na saída!!
- Exemplo:

  A → 65 65 → A

  B → 66 66 → B

  etc…

Codificações de caracteres

- Existem códigos normalizados:
  - ASCII – American Standard Code for Information Interchange (7 bits)
    - Cada símbolo tem um código de 0 a 127

      Exemplo: ´A´ → 1000001_2 = 65

      ´%´→ 0100101_2 = 37
    - inclui códigos de controlo: TAB, RET,…
  - ISO8859 (8bits)
    - mais 127 códigos, suportando caracteres com acentos e outros (depende da variante do código)

      latino 1, … grego, … hebraico, … etc
  - ISO10646 ou Unicode. Múltiplas variantes com número de bits variável. Inclui a representação UTF-8.

Descodificação

- Exemplo: um controlador de ecrã ou de uma impressora recebem sequências de 8 bits e mostram o carácter correspondente
  - exemplo:

    se receberem 01000001_2 fazem aparecer o desenho **A**

    se receberem a sequência

    01010000_2 00010000_2 01010100_2 fazem aparecer:

    2 *(muda de linha)*

    6
  - **NOTA: para escrever números temos de fazer sair os seus códigos de caracteres e não os números!**
  - **Carácter 2 não tem o código 2!**

Descodificação dos bits

- Como interpretar o seguinte grupo de bits?

  01000001

o número 65?

a letra A?

8 valores lógicos (FVFFFFFV)?

outra …??

Todas estas leituras podem ser válidas!

- As linguagens de programação costumam ter tipos de dados para fixar a interpretação

Tipos de dados nas linguagens

- Os tipos são a forma das linguagens ajudarem na interpretação e utilização correta dos dados
- Tipos primitivos

  Em Java e C:

Tipos	Java (norma)	C 32-bit	C 64-bit
`byte`	1	–	–
`char`	2 (unicode)	1	1
`short`	2	2	2
`int`	4	4	4
`long`	8	4	8
`float`	4	4	4
`double`	8	8	8
`boolean`	1bit válido	–	–

No C dependem da arquitetura (os valores apresentados são os típicos).

Estes tipos numéricos são sempre com sinal.

Tipos de dados nas linguagens

- O C oferece ainda:

Tipos	Java	C 32-bit	C 64-bit
`unsigned char`	–	1	1
`unsigned short`	–	2	2
`unsigned int`	–	4	4
`unsigned long`	–	4	8
`long long`	–	8	8
`unsigned long long`	–	8	8
`long double`	–	12	16

No C dependem da arquitetura (os valores apresentados são os típicos).

Operações

- As comuns (Java/C)
  - Aritméticas: +, -, *, /, ...
  - Relacionais: <, >, ==, !=, ...
  - Lógicas: &&, ||, ! *(no C entre inteiros, 0=false)*
  - Lógicas binárias: &, |, ~, >>, <<
- Tipicamente estas operações **entre os tipos primitivos** são diretamente suportadas pelo CPU

Expressões

- O que acontece quando se misturam tipos?
  - Erro de compilação ou promoção (com possível conversão do valor)

```
short i = 1;
double x = 3.14 + i;
```
o que acontece?

- E se o resultado não cabe no tipo original?

```
int i = 50000;
i*i
```
que tipo? Resultado errado?

```
long j = i*i;
```
que tipo? Resultado errado?

```
i*(long)i
```
que tipo? Resultado errado?

Escrevendo estes dados (Java)

- Exemplo `System.out.print(X)` / `println(X)`
  - X é uma String ou então um novo objecto String é construído com uma representação do valor de X
  - String.valueOf(X)
  - Integer.toString(X), Long.toString(X), ...
  - Object.toString()
  - *out* é um Stream (canal) para onde é enviada a representação da String como uma sequência de bytes que é passada ao SO para que este faça aparecer o respectivo texto no écran (ou noutro local)

Em C

- Exemplo **`printf(`***`format, X,...`***`)`**
  - *Format* – **é apenas uma string** que define o que escrever e como apresentar os restantes argumentos

Exemplo: `printf("result: %d, %x, %f\n", 12, 12, 12.0);`

escreve: **`result: 12, c, 12.00000`**↵

- Exemplos de outros formatos:

`%ld` = long em base 10 `%hd` = short em base 10

`%u` = unsigned int base 10 `%lu` = unsigned long base 10

`%e` = double em notação científica

- Em Java também existe `printf()`

Printf faz o indicado pela formatação

- Exemplo:

```
short i = -1;
unsigned short j = i;
int x = j+1;
printf("i=%hd, j=%hu, x=%d", i, j, x);
printf("i=%u, j=%d, x=%f ?=%d", i, j, x);
```

- O compilador pode avisar, mas produz um executável
- Resultado? (pode variar entre arquiteturas....)

```
i=-1, j=65535, x=65536
i=4294967295 j=65535 x=0.00000 ?=134518384
```


C vs Java

- A execução começa sempre na função **`main()`**
- As funções são semelhante aos métodos Java, mas:
  - Não pertencem a objetos: não é preciso criar nada para usar, nem existe `this`
  - São como os métodos `static`
  - Não há polimorfismo (um nome = uma função)

C vs Java

- Exemplo de declaração completa do main:

```
int main(int argc, char *argv[]) {
    printf("hello world\n");
    return 0;
}

public static void main(String[]args){
    System.out.println("hello world");
}
```


C vs Java

- Comentários:

```
/* bloco
 */

// linha
```

- Blocos de código e escopo das declarações:

```
{ declarações;
  instruções;
}
```


C vs Java

- Construções de controlo:

```
if (exp) {  inst;
} else {  inst;
}

while ( exp ) {
  inst;
}

for ( inst; exp; inst ) {
  inst;
}
```

```
switch (expr) {
  case val: inst; break;
  ....
  default: inst;
}

do {
   inst;
} while (exp);
```


C tem pre-processador

- Diretivas iniciadas por #
  - `#define XPTO 234` - no ficheiro, substitui XPTO por 234
  - Permite pseudo-constantes, e mais...
  - `#include "file.h"` - inclui o file.h da diretoria corrente
  - `#include <file.h>` - inclui o file.h, da diretoria standard
  - Permite introduzir as definições de funções e tipos fora do nosso ficheiro como das bibliotecas (interfaces)
  - Também chamados ficheiros de headers

Exemplo completo (hello.c)

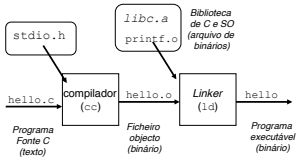
```
#include <stdio.h>

int main(int argc, char *argv[]) {
    // escreve 5 linhas
    for (int i=0; i<5; i++)
        printf("hello world\n");
    return 0;
}
```


Processo de compilação de C

- compilação do programa:

```
cc -o hello hello.c
```



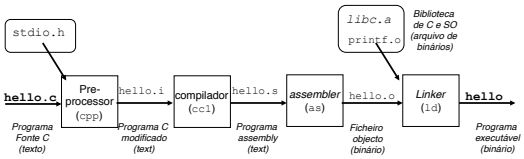
- Existem erros e *warnings*
  - de compilação e de ligação

Processo de compilação de C

- compilação do programa:

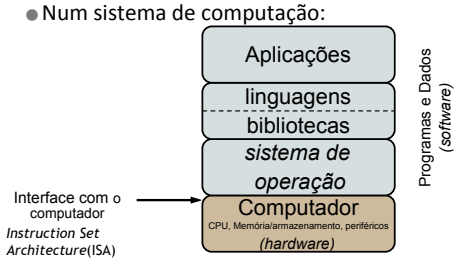
```
cc –o hello hello.c
```

- Os compiladores podem ter mais passos intermédios:



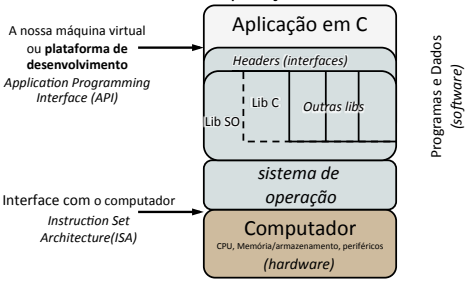
Níveis nos sistemas informáticos

- Num sistema de computação:



Níveis nos sistemas informáticos

- Num sistema de computação com C:



C vs Java

Java	**C**
object-oriented	function-oriented
strongly-typed	can be overridden
method overloading	no function overloading
classes for name space	single name space (mostly), file-oriented
exceptions, exception handling	if (f() < 0) {error} or OS signals
String is a type	Just char[], with '\0' end
Standard classes lib	Standard functions lib