

Arquitectura de Computadores

Aulas prácticas

© 2009 Pedro Freire

Este documento tem alguns direitos reservados:



Atribuição-Uso Não-Comercial-Não a Obras Derivadas 2.5 Portugal
<http://creativecommons.org/licenses/by-nc-nd/2.5/pt/>

Isto significa que podes usá-lo para fins de estudo.

Para outras utilizações, lê a licença completa. Crédito ao autor deve incluir o nome ("Pedro Freire") e referência a "www.pedrofreire.com".

| | |
|--|-----------|
| REQUISITOS | 6 |
| PARA A ESCOLA | 6 |
| PARA O ALUNO | 6 |
| AULA 01 | 7 |
| AMBIENTE DE TRABALHO..... | 7 |
| MONTAR/COMPILAR E CORRER..... | 7 |
| EXPLICAR O PRIMEIRO EXEMPLO | 8 |
| HARDWARE: CPU, RAM E ENDEREÇAMENTO | 9 |
| AULA 02 | 11 |
| CONSTANTES..... | 11 |
| CARACTERES E <i>STRINGS</i> | 11 |
| ETIQUETAS (<i>LABELS</i>) E VARIÁVEIS..... | 12 |
| DB, DW, DD, DQ | 12 |
| EQU E \$..... | 14 |
| AULA 03 | 14 |
| REGISTOS..... | 14 |
| <i>FLAGS</i> | 14 |
| SEGMENTOS/SECÇÕES | 14 |
| EXERCÍCIO..... | 14 |
| AULA 04 | 14 |
| NOME.ASM..... | 14 |
| RESB, RESW, RESD, RESQ..... | 14 |
| LER DADOS DO TECLADO..... | 14 |
| REFERÊNCIAS À MEMÓRIA..... | 14 |
| EXERCÍCIO..... | 14 |
| AULA 05 | 14 |
| PALAVRAS.ASM..... | 14 |
| EXERCÍCIOS..... | 14 |
| AULA 06 | 14 |
| A06-CONVERTE2.ASM..... | 14 |
| EXERCÍCIOS..... | 14 |
| AULA 07 | 14 |
| A07-CONVERTE2A.ASM..... | 14 |
| PILHA | 14 |
| EXEMPLO..... | 14 |
| PASSAR ARGUMENTOS A SUB-ROTINAS/FUNÇÕES USANDO A PILHA | 14 |
| EXERCÍCIO..... | 14 |
| AULA 08 | 14 |
| LEITURA DE FICHEIROS: A08-LE2.ASM | 14 |
| ABRIR/FECHAR FICHEIROS..... | 14 |

| | |
|--|-----------|
| <i>HANDLE</i> (“MAÇANETA” OU NÚMERO DE FICHEIRO) | 14 |
| ASCIIZ | 14 |
| ESCRITA PARA FICHEIROS: A08-ESCR.ASM | 14 |
| EXERCÍCIO..... | 14 |
| AULA 09 | 14 |
| OPERAÇÕES ARITMÉTICAS: A09-CALCULADORA.ASM..... | 14 |
| DETECÇÃO DE EXCESSO (OVERFLOW) | 14 |
| MÚLTIPLA PRECISÃO..... | 14 |
| EXERCÍCIOS..... | 14 |
| AULA 10 | 14 |
| OPERAÇÕES EM INTEIROS COM SINAL | 14 |
| DETECÇÃO DE EXCESSO (OVERFLOW) | 14 |
| MÚLTIPLA PRECISÃO..... | 14 |
| EXERCÍCIO..... | 14 |
| AULA 11 | 14 |
| OPERAÇÕES EM NÚMEROS FRACCIONÁRIOS | 14 |
| EXERCÍCIO..... | 14 |
| AULA 12 | 14 |
| EXERCÍCIOS | 14 |
| VIRTUALBOX LINUX..... | 14 |
| INSTALAR O SOFTWARE BASE | 14 |
| INSTALAR A IMAGEM DO XUBUNTU | 14 |
| CRIAR A MÁQUINA VIRTUAL DO XUBUNTU..... | 14 |
| USAR O XUBUNTU..... | 14 |
| BIBLIOGRAFIA..... | 14 |

Requisitos

Para a escola

Requisitos para as salas das aulas práticas de Arquitectura de Computadores:

- Um qualquer Linux instalado, desde que suporte o NASM (ver www.nasm.us), nativo, em *dual-boot* ou máquina virtual (VirtualPC ou VMware). Pode ainda ser acedido remotamente via Telnet/SSH desde que todo o processo de aceder ao sistema seja de forma visual em *single-click* até ao surgimento da *prompt* utilizador+senha.
- NASM (*Netwide Assembler*) instalado no Linux, e assegurar a existência do *linker* (ld).
- Assegurar que o Linux tem ambiente visual instalado de fácil acesso (par utilizador+senha), assim como editor de texto visual (à semelhança do Bloco de Notas do Windows).
- Acesso à Internet com um *browser*.

Deve haver 1 PC por aluno.

Cada aula está programada para uma duração de 1,5h.

Para o aluno

Comparência nas aulas. Este guião tem propositadamente omissos certos elementos importantes para a compreensão total da matéria (notas históricas, relações entre partes diferentes da matéria, avisos sobre erros comuns, etc., ou seja, elementos para uma nota 20), embora seja suficiente para passar com nota bastante acima de 10.

Deves ter instalado o Linux em computador próprio se quiseres acompanhar a matéria em casa. Consulta www.pedrofreire.com para uma pequena introdução de como o fazer. O professor irá distribuir uma cópia de um Linux já preparado para as aulas aos alunos que tiverem dificuldade em instalá-lo sozinhos: pede a cópia ao professor e consulta a secção “VirtualBox Linux” ao fim deste Guião. Vê também a secção acima para requisitos. Não é no entanto de todo necessário que tenhas estes sistemas em casa para conseguires passar à cadeira (podes usá-los na escola).

Esta cadeira assume que já tens experiência no uso de computadores (não necessariamente no Linux).

Aula 01

Introdução e contextualização: o Assembly, o nasm e Linux. Hiperligações.
Referências e avaliação.

Ambiente de trabalho

Os exercícios desta cadeira estão feitos para funcionarem sobre Linux. Vê em www.pedrofreire.com as tuas opções para instalar este sistema no teu computador. Existe uma pequena introdução ao final deste Guião para instalar um Linux fornecido pelo professor (secção “VirtualBox Linux”, antes da Bibliografia).

Qualquer versão de Linux serve. Tens no entanto de instalar o nasm no teu. Consulta www.nasm.us ou pesquisa na Internet por uma versão rpm do nasm que será mais fácil de instalar na maior parte dos Linux.

Quando tiveres o Linux instalado localiza a consola/terminal (“linha de comandos” do Linux) e ambienta-te ao sistema de directórios, *browser* e editor de texto. Ambienta-te também aos comandos `cd`, `ls`, `cp`, `mv` e `rm` básicos na linha de comandos.

O comando

```
man comando
```

dá-te ajuda sobre o comando `comando`.

Montar/compilar e correr

Descarrega o primeiro exemplo (`hello.asm`). Este é o primeiro programa típico de qualquer linguagem: exhibe a frase “Olá, mundo!” (“*Hello, world!*”) no ecrã.

Para o experimentarmos vamos precisar de o montar (“*assemble*” – outras linguagens usam a expressão “compilar”).

Para montar qualquer programa `prog.asm` genérico, precisamos de correr os seguintes comandos na consola/terminal:

```
nasm -f elf prog.asm
ld prog.o
./a.out
```

As partes a cor mais clara referem-se a este exemplo em particular, e devem ser alteradas em cada programa que se vai montar. Por exemplo, no nosso `hello.asm`, substituímos `prog.asm` por `hello.asm` e `prog.o` por `hello.o`.

Em algumas versões de Linux mais recentes, será necessário usar “elf64” no primeiro comando, em vez de “elf”. Testa e experimenta a ver qual funciona para ti. Nota que se precisas de “elf64” não vais conseguir correr alguns exemplos da cadeia.

Os dois primeiros comandos não irão exibir nada se tudo correr bem. O último comando corre o programa. No caso do `hello.asm`, isto simplesmente exibe no ecrã:

```
Hello, world!
```

Explicar o primeiro exemplo

Abre o `hello.asm`. Ele pode ser aberto com o Bloco de Notas (*Notepad*) no Windows, e deve abrir automaticamente com um duplo-clique em Unix (caso contrário abre-o num editor de texto qualquer: `gedit`, `kwrite`, `kate`, `vi`, etc.).

```
section .data
msg     db     "Hello, world!",0xa
len     equ   $ - msg

section .text

    global _start

_start:

;write our string to stdout

        mov     edx,len
        mov     ecx,msg
        mov     ebx,1
        mov     eax,4
        int     0x80

;and exit

        mov     ebx,0
        mov     eax,1
        int     0x80
```

Legenda:

- **Comentários** – texto descritivo para o programador documentar o código. É ignorado pelo `nasm` e pelo computador (não fará sequer parte do programa final). Começam com um “;” numa linha e são comentários até ao final dessa linha. Vários comentários do ficheiro `hello.asm` foram eliminados na versão exibida acima.
- **Directivas** – comandos que dizem ao `nasm` como organizar certas partes do programa. Veremos estas em mais detalhe em aula posterior.
- **Instruções** – Código real que será executado pelo computador.

As instruções têm um nome, seguido de zero, um ou dois argumentos separados por vírgula. Este programa só usa duas instruções: `mov` (abreviatura de *move*) e `int` (abreviatura de *interrupt*). `mov` copia valores do seu segundo argumento para o primeiro e então

```
mov edx, len
```

equivale (esquemáticamente) a

```
edx = len
```

`int 0x80` por seu lado é usado no Linux para “chamar” o próprio Linux, i.e., para lhe pedir para fazer algum trabalho em nome da aplicação.

Cada um dos dois blocos de código (“*write our string to stdout*” e “*and exit*”) explicados na aula pede uma funcionalidade diferente ao Linux. O primeiro pede para se escrever uma mensagem no ecrã e o segundo termina o programa.

Hardware: CPU, RAM e endereçamento

O *Assembly* é a única linguagem que é directamente entendida pelo próprio *hardware* (i.e., pelos circuitos que compõem um computador). Dois destes circuitos são de extrema importância:



Processador central
(CPU)



Memória (RAM)

O processador central é o “cérebro” do computador. Ele é quem reconhece as instruções e age de acordo com elas. A memória é um grande “bloco de notas” do processador: guarda informação.

A memória hoje em dia tem milhares de milhões de bytes (gigabytes). Precisamos de poder referir cada um desses bytes de forma independente. Precisamos de poder “apontar” para um e dizer ao CPU: “escreve naquele ali”.

Isso faz-se usando endereçamento, tal como temos moradas para entrega de correio dos CTT. No computador, cada “morada” de um byte é um número inteiro de 0 em diante.

Tudo o que temos na memória é então um “fio” de números guardados em cada byte. Se esse número se refere a uma música, a um vídeo, a uma carta, ou a um programa, só depende de como está a ser usado em cada momento. Quem dá vida a um computador é o software. Sem ele, o computador não é mais do que uma máquina de calcular cara.

Um último conceito importante sobre a memória é que todos os seus bytes têm sempre um valor numérico qualquer. Eu posso não saber de antemão que valor é

esse (em cujo caso digo que ele é desconhecido ou “lixo”), mas ele existe sempre. Não há o conceito de um byte “não ter nada”.

Aula 02

Explicação de vários elementos do `hello.asm`.

Constantes

Em várias partes do nosso primeiro exemplo vemos constantes. O `nasm` suporta vários tipos de constantes:

- Numéricas decimais (e.g.: 75)
- Numéricas hexadecimais (e.g.: `0x4B`, `04Bh` ou `$04B`)
- Numéricas octais (e.g.: `113q` ou `113o`)
- Numéricas binárias (e.g.: `1001011b`)
- Numéricas de vírgula flutuante, em decimal (e.g.: `75.0`)
- Caracteres (e.g.: `'K'`)
- *Strings* (cadeias/seqüências de caracteres) (e.g.: `"Letra K"`)

Exemplos e detalhes explicados na aula.

Exercício: Identifique as 8 constantes usadas no `hello.asm` e o seu respectivo tipo.

Caracteres e *strings*

Existe uma norma universal para correspondência entre números e caracteres ocidentais: o ASCII. Podes ver a tabela ASCII em

<http://pt.wikipedia.org/wiki/ASCII>.

Como podes ver, apenas estão definidos 128 códigos ASCII. Isto cabe num byte. Então por razões práticas, e para a totalidade de exercícios da cadeira, podemos assumir que 1 caracter = 1 byte.

Dos 33 caracteres de controle, dois são importantes para a cadeira: o CR (*Carriage Return*, ou “retorno do carro”), código `0xD`, e o LF (*Line Feed*, ou “alimentar linha”), código `0xA`. Dependendo do sistema operativo onde estamos a trabalhar, diferentes combinações destes caracteres compõem uma quebra de linha (EOL):

| | |
|---------------------------|-----------------------|
| Windows: | <code>0xD, 0xA</code> |
| Unix (incluindo Linux): | <code>0xA</code> |
| Apple Macintosh (antigo): | <code>0xD</code> |

Ver também http://pt.wikipedia.org/wiki/Nova_linha.

Uma *string* (cadeira/sequência de caracteres) não é mais do que vários caracteres escritos uns a seguir aos outros em memória.

Etiquetas (*labels*) e variáveis

No `hello.asm` vemos 3 etiquetas ao todo: `msg`, `len` e `_start`.

Uma etiqueta identifica uma zona do código. Se precede uma instrução, deve ser seguida de dois pontos ("`:`"), como a `_start`. Quando usada por si mesma no código, ela representa o endereço de memória da zona que identifica.

Quando usada indirectamente para se aceder à zona de memória que identifica, como em:

```
[msg]
```

podemos também chamar a essa etiqueta e à zona de memória que identifica de "variável" (como uma variável matemática). Mais sobre este uso em aula posterior. No `hello.asm`, podemos chamar a `msg` de variável.

Quando usada com a directiva `equ` (ver abaixo) podemos também chamar a essa etiqueta de constante (por si mesma). No `hello.asm`, podemos chamar a `len` de constante.

db, dw, dd, dq

Estas são as directivas "*define*" (definir). Definem números de tamanhos diferentes, dependendo da segunda letra da directiva:

| | <i>Define...</i> | Bits | Bytes | Máximo valor |
|----|---------------------|------|-------|-----------------------|
| db | Bytes | 8 | 1 | 255 |
| dw | <i>Words</i> | 16 | 2 | 65535 |
| dd | <i>Double-words</i> | 32 | 4 | ≈4 mil milhões |
| dq | <i>Quad-words</i> | 64 | 8 | ≈1,8×10 ¹⁹ |

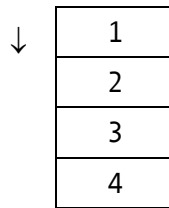
As directivas *define* criam espaço na memória do nosso programa com os dados especificados à frente deles (números, caracteres ou *strings* separados por vírgulas).

Cada número especifica um byte, *word*, etc., diferente, dependendo da directiva.

Exemplo:

```
db 1, 2, 3, 4
```

Esta directiva cria em memória os seguintes bytes:



Trivial. Mas, e esta directiva?

```
dw 1, 2, 3, 4
```

Bem, vamos ver. Como se representa o número 1 em binário? 1. E se eu quiser que ele ocupe um byte inteiro (8 bits)? Adiciono zeros à esquerda:

00000001

E se eu quiser que este número tenha 16 bits? Continuo a adicionar zeros à esquerda:

00000000 00000001

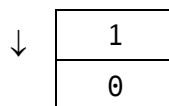
Este número de 16 bits tem agora o tamanho de uma *word*. Podemos dividi-lo nos seus dois bytes componentes e reconverter para decimal:

00000000 = 0; 00000001 = 1

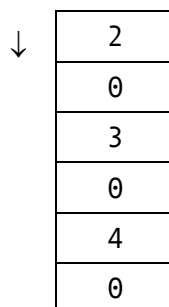
Os processadores Intel que estudamos guardam estes bytes em memória da direita para a esquerda (bytes menos significativos primeiro) – formato *little-endian*: ver

http://pt.wikipedia.org/wiki/Little_endian

Então o seu aspecto em memória vai ser:



Repetindo o processo para as restantes *words* do exemplo, ficamos com:



Seguindo o mesmo raciocínio, com esta directiva:

```
dd 1, 2
```

Vamos ver:

| | |
|---|---|
| ↓ | 1 |
| | 0 |
| | 0 |
| | 0 |
| | 2 |
| | 0 |
| | 0 |
| | 0 |

equ e \$

A directiva **equ** é a abreviatura de *equate* e significa “igualar a”. Torna a etiqueta à sua esquerda uma constante por si mesma, com valor igual ao resultado da expressão que segue o **equ**.

Exemplos:

```
PI          equ 3.141592653589793238462
PI_x_2     equ PI * 2
```

Por convenção, as etiquetas das constantes costumam estar escritas em maiúsculas.

Um símbolo especial que pode ser usado nas expressões **equ** e noutra qualquer sítio onde seja aceite um inteiro é o **\$**. Ele representa o “endereço inicial da linha onde está o **\$**”, ou o “próximo endereço disponível” se esta não gerar código.

É com recurso ao **\$** que o comprimento da *string* “*Hello, world!*” está a ser calculado no `hello.asm`. Mais detalhes na aula.

Exemplos:

```
; Se aqui $=0, então...
                db    1, 2
A               equ   $           ; A = 2
                dw    3
B               equ   $           ; B = 4
```

Aula 03

Todas as instruções `mov` do `hello.asm` usam registos e estão no “segmento de código”.

Registos

Registos não são mais do que pequenas memórias internas do processador, usadas para cálculos temporários. Cada uma dessas memórias tem um nome diferente, tem um tamanho de 32-bits e guarda um número inteiro.

Os registos mais comuns e que vamos usar nesta cadeira são:

- `eax` (Acumulador)
- `ebx` (Base)
- `ecx` (Contador)
- `edx` (Dados)

- `esi` (Endereço de origem – *Source Index*)
- `edi` (Endereço de destino – *Destination Index*)
- `ebp` (Endereço base – *Base Pointer*)
- `cs` (Segmento de código)
- `ds` (Segmento de dados)
- `es` (Segmento extra)
- `ss` (Segmento da pilha – *Stack Segment*)

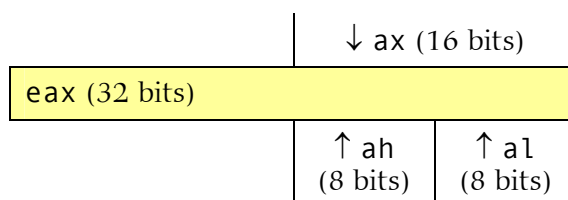
Os primeiros 4 são os registos de uso geral. Os restantes têm comportamentos específicos com certas instruções que veremos mais tarde. Os detalhes da utilização de todos estes registos e do nome dado a cada um (acumulador, etc.) é explicado na aula.

Podemos escrever um valor num registo com a instrução `mov`:

```
mov  eax, 420
```

Os registos podem ser lidos da mesma forma, e usados em quase todas as restantes instruções do processador.

Os registos de uso geral têm outros nomes que se referem a partes do mesmo registo. Por exemplo, para o `eax`:



À metade inferior do `eax`, chamamos de `ax`. Esta subdivide-se na metade alta (*high*) com o nome `ah`, e a metade baixa (*low*) com o nome `al`.

Lembra-te no entanto que estes são nomes para partes diferentes do mesmo registo!

Exercício resolvido:

```
mov  eax, 0xABCD1234
mov  ax,  0xFFFF
```

Estas instruções são válidas? Quantos bits têm cada um dos seus argumentos?*

Depois destas duas linhas, qual o valor de `eax`?[†] Porquê?

Agora acrescento ainda esta linha:

```
mov  ah,  0x00
```

Depois das três linhas, qual o valor de `eax`?[‡] Porquê?

Os registos `esi`, `edi` e `ebp` não têm versões de 8 bits. Os registos de segmentos são explicados abaixo.

Flags

O registo `flags` é um registo especial, de 32 bits. Não podemos aceder a este registo directamente, mas sim a alguns dos seus bits, em instruções especializadas. Isto porque cada um dos seus bits representa uma *flag* (bandeira ou bandeirola, mas vamos continuar a usar o estrangeirismo porque é mais comum).

Uma *flag* é um bit que representa um estado de um resultado de uma operação, ou um estado indicador para o processador se comportar de certa maneira em certas situações. Cada *flag* é representada por uma única letra. Vamos ver três exemplos comuns.

A *flag* Zero (**Z**) indica se a última operação aritmética teve um resultado de zero. Se (por exemplo) fizermos a conta 5-5 em Assembly, a *flag* Z fica activa (a 1). Podemos depois testar essa *flag* com as instruções de salto condicional `JZ` e `JNZ` (em detalhe em aula posterior).

A *flag* Carry (**C**) indica se a última operação aritmética teve *carry* (transporte) no último bit (do 32º para o 33º no caso de operações de 32 bits). Se (por exemplo) somarmos 3 mil milhões a 3 mil milhões em Assembly, a *flag* C fica activa (a 1). Podemos depois testar essa *flag* com as instruções de salto condicional `JC` e `JNC` (em detalhe em aula posterior).

* `eax` e `0xABCD1234` têm ambos 32 bits, logo a instrução é válida. `ax` e `0xFFFF` têm ambos 16 bits pelo que a instrução também é válida.

† `0xABCDFFFF`

‡ `0xABCD00FF`

A *flag* de Direcção (D) indica ao processador se as instruções de *string* devem varrer sequências de bytes na direcção normal (aumentando o endereço, quando D=0, o que se faz com *CLD*, *Clear Direction flag*) ou na direcção inversa (diminuindo o endereço, quando D=1, o que se faz com *STD*, *Set Direction flag*). Mais detalhes sobre instruções de *string* em aula posterior.

Segmentos/secções

Segmentação é uma estratégia para resolver dois problemas: *relocation* (mudança de endereço) e protecção de código.

Imagina que ligaste agora o computador. Corres o `hello.asm`. Porque há poucos programas a correr, ele é carregado talvez no endereço 250 mil. E então o endereço da mensagem "*Hello, world!*" é (hipoteticamente) `250050`.

Mas agora lêes notícias da Internet enquanto abres um filme de DVD e descarregas e-mail. Voltas a correr o `hello.asm` que fica no endereço 710 mil. O endereço da mensagem "*Hello, world!*" é (hipoteticamente) `710050`.

Então... que endereço deve o `nasm` colocar na linha

```
mov     ecx,msg
```

para representar a mensagem?

A segmentação resolve este problema com registos que mantêm o endereço onde as várias secções do programa foram colocadas:

- Código (CS)
- Dados inicializados (DS e ES)
- Dados não inicializados (DS e ES)
- Pilha (SS)

Em qualquer acesso à memória, agora o processador faz automaticamente a soma desses endereços iniciais com o endereço que se está a pedir no código. Assim o `nasm` só tem de assumir que está num "mundo perfeito" onde as secções de qualquer programa começam sempre no endereço zero (0).

Claro que para isto se conseguir, o programador tem de identificar no seu código a que secções pertence cada instrução e directiva. Isto faz-se com a directiva `section` que pode levar os seguintes argumentos:

- `.text` (secção/segmento de código)
- `.data` (secção/segmento de dados inicializados)
- `.bss` (secção/segmento de dados não inicializados)

A pilha é criada implicitamente.

A protecção de código tem a ver com assegurar-se que o nosso programa trabalha apenas nas zonas de memória que lhe foram reservadas e não fora delas, destruindo outros programas. O MS-DOS antigo e o Windows anterior ao

Windows NT e Windows 95 não usava estes mecanismos de protecção pelo que um único programa com erros poderia trazer abaixo todo o sistema.

Hoje em dia o Windows (e desde sempre, o Unix) usam as capacidades dos processadores de verificarem em cada acesso à memória, se esse acesso está dentro dos limites conhecidos de cada segmento, e apenas permitem o acesso nessas condições.

Por esta razão os registos de segmento (CS, DS, etc.) hoje em dia não são meros endereços iniciais das partes dos programas, mas coisas mais complexas e com mais informação. Por essa razão não se deve alterar os registos de segmento, excepto para copiar um para outro, e.g.:

```
mov  eax, ds
mov  es,  eax
```

Exercício

Isto conclui a nossa análise ao `hello.asm`. Deves agora ser capaz de compreender todo o código deste programa em detalhe. Se isso não for verdade, consulta as aulas anteriores.

Para testar os teus conhecimentos vamos tentar dois exercícios.

1. Altera o `hello.asm` para que em vez de ele dizer “Olá, mundo!”, diga “Adeus mundo cruel!”.
2. Altera o `hello.asm` para que ele exiba ambas as frases ao mesmo tempo (uma em cada linha). Faz isto com duas versões: uma alterando só a secção de dados, e outra que também acrescenta um novo bloco para exibir uma nova frase.

Correcção na aula.

Aula 04

Hoje vamos resolver um novo programa: o `nome.asm`. Monta-o e corre-o. O que faz?

nome.asm

Este programa tem algumas coisas novas. Começamos por:

```
section .bss
nome    resb    MAX_NOME
```

Na secção de dados não inicializados (`.bss`) estamos a definir uma etiqueta `nome` com recurso à directiva `resb`. Esta é uma nova directiva.

resb, resw, resd, resq

`resb` é a abreviatura de *Reserve Bytes* (reservar bytes). Recebe um único argumento que será a quantidade de bytes a reservar. Existem variantes da instrução para outros tamanhos de reserva:

| <i>Reserve...</i> | |
|-------------------|---------------------|
| <code>resb</code> | <i>Bytes</i> |
| <code>resw</code> | <i>Words</i> |
| <code>resd</code> | <i>Double-words</i> |
| <code>resq</code> | <i>Quad-words</i> |

Estas instruções reservam espaço suficiente para guardar a quantidade pedida de bytes, *words*, etc.. Estes dados não estarão inicializados, pelo que pertencem à secção `.bss`. Compare isto com os `db`, `dw`, etc., onde especificamos o valor inicial de cada elemento, daí que sejam dados inicializados (secção `.data`).

Exemplo:

```
db 5
resb 5
```

Quantos bytes estão definidos em cada linha? E qual é o valor inicial de cada byte?*

```
dw 1
resw 2
```

Quantos bytes estão definidos em cada linha? E qual é o valor inicial de cada byte?†

Ler dados do teclado

Do restante código, só um novo bloco “Lê nome” é novo:

```
; Lê nome
mov     edx, MAX_NOME
mov     ecx, nome
mov     ebx, 0
mov     eax, 3
int     0x80
mov     [nome_len], eax
```

Este bloco, explicado na aula, só tem uma instrução de aspecto novo: a última.

Referências à memória

Sempre que usamos numa instrução os parêntesis rectos, estamos a tentar ler/escrever na memória principal (RAM) do sistema. O valor que estiver dentro dos parêntesis rectos (que deve ser um número inteiro, uma etiqueta, ou um registo de 32 bits) é o endereço a partir do qual desejo ler/escrever.

Então, neste exemplo, se `nome_len` é o endereço da *double-word* `0`, então `[nome_len]` lê/escreve em cima do `0`. É por esta razão que uma etiqueta com parêntesis rectos à volta se pode chamar de “variável” (ver aula anterior).

Como a instrução `mov` copia sempre os valores da direita para a esquerda, a última linha do bloco acima:

```
mov [nome_len], eax
```

escreve `eax` na variável `nome_len`.

* `db 5` define 1 byte com o valor 5; `resb 5` define 5 bytes com valor indefinido (desconhecido).

† `dw 1` define 2 bytes com os valores 1 e 0 (ver aula anterior); `resw 2` define 4 bytes com valor indefinido (desconhecido).

Então obviamente

```
mov eax, [nome_len]
```

será então a forma de ler uma variável.

Exercício

Faz com que o programa responda não apenas “Bem-vindo *nome*”, mas “Bem-vindo *nome* a este PC”.

Correcção na próxima aula.

Aula 05

Resolução do exercício da aula anterior.

Para exibir mais uma frase vamos precisar de mais um bloco para escrever para o ecrã. Como este bloco se vai referir a uma frase estática, vamos copiar (por exemplo) o bloco “Escreve o cumprimento”, inserindo-o na posição certa do código para ter o comportamento que desejamos no ecrã: depois de “Escreve o nome”, mas antes do “fim”:

```
; Escreve "a este PC"
    mov     edx, glen
    mov     ecx, greet
    mov     ebx, 1
    mov     eax, 4
    int     0x80
```

As letras mais claras referem-se às partes que copiámos do bloco “Escreve o cumprimento” e que foram modificadas.

No entanto este novo bloco, tal como está, escreve a frase `greet` (“Bem-vindo”) no ecrã. Queremos que ele escreva outra frase que ainda não foi definida ao topo do código. Então temos de a definir, por exemplo logo abaixo de `greet` e `glen`:

```
greet2 db " a este PC!"
g2len  equ $ - greet2
```

As letras mais claras referem-se às partes que copiámos de `greet` e `glen` e que foram modificadas.

Note-se que demos um novo nome a esta nova *string*: `greet2`. O comprimento desta *string* também teve de ter um novo nome: `g2len`.

Aplicamos então estes novos nomes no novo bloco que adicionámos ao fim do código:

```
; Escreve "a este PC"
    mov     edx, g2len
    mov     ecx, greet2
    mov     ebx, 1
    mov     eax, 4
    int     0x80
```

As letras mais claras referem-se às alterações.

Monta e corre. Verificas dois problemas:

1. A frase surge na linha abaixo do nome
2. A linha de comandos fica à frente da frase, e não por baixo.

Resolver o ponto 2 é mais fácil. O que falta à frase é uma quebra de linha ao final. Tal como no `hello.asm`, isso resolve-se acrescentando o código `0xA` ao final:

```
greet2 db " a este PC!", 0xA
```

Monta e corre. Este problema foi resolvido.

O primeiro problema é mais estranho. O que acontece é que se pedir ao Linux para ler texto do teclado e a pessoa escreve “Ana” seguido de *Enter* (↵), o Linux devolve no registo `eax` o valor 4 indicando que foram lidos 4 caracteres: “A”, “n”, “a” e o *Enter* em si (`0xA`)!

Agora precisamos de eliminar esse `0xA`. Como ele é sempre o último carácter lido, a solução é fácil: em vez de dar ao Linux o tamanho 4 para escrever para o ecrã, damos o tamanho 3, efectivamente eliminando o `0xA` da lista de caracteres a escrever para o ecrã!

Como o `0xA` é sempre o último carácter, só temos sempre de subtrair 1 ao valor de `eax` recebido antes de enviar a frase para o ecrã. Isto pode ser feito ao fim do bloco “Lê nome” (antes de escrever em `nome_len`), ou ao início do bloco “Escreve o nome” (depois de ler `nome_len`). Vamos optar pela primeira e adicionar esta linha antes da última do bloco “Lê nome”:

```
sub    eax, 1
```

`sub` é uma instrução nova que faz uma subtracção. Mais detalhes abaixo.

Monta e corre. Todos os problemas foram resolvidos.

Questão: como farias a alteração se ela fosse no bloco “Escreve o nome”?

palavras.asm

Monta e corre este programa. O que faz ele?

Este programa usa várias novas instruções novas para trabalhar. Elas (e outras da mesma família) são as seguintes:

| Operações aritméticas | |
|-----------------------|--|
| <code>add</code> | Adição. Tal como um <code>mov</code> , recebe dois argumentos, soma os dois, e guarda o resultado no primeiro. |
| <code>sub</code> | Subtracção – ver <code>add</code> . |
| <code>inc</code> | Incrementar (i.e., somar 1). Recebe apenas um argumento (que é incrementado). |
| <code>dec</code> | Decrementar (subtrair 1) – ver <code>inc</code> . |

Operações de *string*

lods *S* *Load string* (carregar a partir da *string*). Não tem argumentos. Dependendo de *S*, carrega um número da memória no endereço em *esi* para *al*, *ax* ou *eax*, e soma (subtrai se a *flag D* for 1) um valor adequado a *esi*:

| <i>S</i> | Registo | <i>esi</i> ± ... |
|----------|------------|------------------|
| b | <i>al</i> | 1 |
| w | <i>ax</i> | 2 |
| d | <i>eax</i> | 4 |

Exemplo:

```
section .data
dados    db    1, 2, 3
section .text
        mov   esi, dados
        cld
        lods
```

al terá agora o valor 1, e *esi* terá o endereço do 2.

stos *S* *Store string, move string e compare string*. Operações alternativas de *strings* explicadas em aulas posteriores.
movs *S*
cmps *S*

Operações condicionais (comparações e saltos)

cmp Comparar. Faz o mesmo que *sub*, mas não guarda o resultado: altera apenas as *flags* de acordo com o resultado. Ver *sub*.

jmp *Jump* (salto incondicional). O CPU executa a próxima instrução na localização marcada pela etiqueta (*label*) que segue o *jmp* como argumento.

jc *Jump if ...* (salto condicional). O CPU só executa o salto se a condição *c* se verifica. A condição é verificada simplesmente olhando para as *flags* e o seu nome faz sentido se colocado entre os dois argumentos de um *cmp* (ou *sub*) que o preceda.

c pode ser:

| | | |
|----|------------------------|---|
| e | <i>Equal</i> | = |
| ne | <i>Not Equal</i> | ≠ |
| a | <i>Above</i> | > |
| ae | <i>Above or Equal</i> | ≥ |
| b | <i>Bellow</i> | < |
| be | <i>Bellow or Equal</i> | ≤ |

Exemplo:

```
mov eax, 5
cmp eax, 8
jae destino
```

O salto não é executado porque $5 \geq 8$ é falso.

loop *Loop* (ciclo). Decrementa `ecx`. Se o resultado após a decrementação não for zero, salta para a etiqueta (*label*) que segue o `loop`.

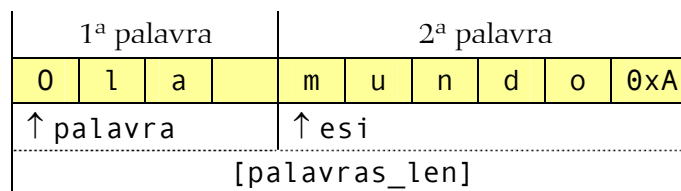
Agora explicar o código torna-se mais fácil.

A secção de dados não tem nada de novo.

A secção de código começa com três blocos conhecidos (escrever frase, ler frase e escrever nova frase). O último bloco também é o nosso habitual para terminar o programa. Os blocos do meio fazem o trabalho de exibir as duas palavras pela ordem inversa.

O raciocínio (explicado em detalhe na aula) é simples: procura-se pelo carácter espaço que separa as duas palavras introduzidas. Depois é só fazer contas para apenas exibir a segunda palavra (bloco “Escreve a segunda palavra”, aproveitando o `0xA` desta para a separar da seguinte) e apenas escrever a seguir a primeira palavra (bloco “Escreve a primeira palavra”).

Esquemáticamente, após o bloco “Procura espaço” terminar, vamos ter (se você escrever “Ola mundo” quando corre o programa):



Exercícios

Altere o programa `palavra.asm` para:

1. Limpa o texto exibido pelo programa. Deseja-se que as duas palavras surjam trocadas na mesma linha, separadas por um espaço, com a linha de comandos a surgir por debaixo delas.
2. Detectar quando só se escreveu uma palavra. Nesse caso exibe uma mensagem de erro no ecrã e volta a pedir duas palavras.

Dicas:

1. Podemos escrever caracteres adicionais para o ecrã criando blocos que só escrevem 1 carácter para o ecrã. Mas nota que à frente de cada palavra está um carácter que não preciso e posso alterar antes de enviar a frase para o ecrã...
2. Detectamos que só se escreveu uma palavra se não havia nenhum espaço nos dados. Descubra em que parte do código ele passa nessa situação (e apenas nessa). É aí que tem de adicionar novo bloco de escrever (a mensagem de erro) e saltar para o bloco de `inicio`.

Aula 06

Correcção do exercício da aula anterior, na aula.

Hoje vamos resolver um novo programa: o `a06-converte2.asm`. Monta-o e corre-o. O que faz?

a06-converte2.asm

Este programa usa várias novas instruções novas para trabalhar. Elas (e outras da mesma família) são as seguintes:

| Operações <i>bitwise</i> (bit a bit) | |
|--------------------------------------|--|
| <code>and</code> | “e” lógico, bit a bit. Tal como um <code>mov</code> , recebe dois argumentos, faz “e” lógico entre os dois, e guarda o resultado no primeiro. |
| <code>or</code> | “ou” lógico, bit a bit – ver <code>and</code> . |
| <code>xor</code> | “ou exclusivo” lógico, bit a bit – ver <code>and</code> . Uma propriedade interessante desta operação lógica, é que quando os dois argumentos são o mesmo, o resultado é sempre zero. Ou seja: <code>xor eax, eax</code> na realidade simplesmente coloca 0 em <code>eax</code> . |
| Deslocamentos e rotações | |
| <code>shl</code> | <i>Shift left</i> (deslocamento à esquerda). Move todos os bits do 1º argumento tantos bits para a esquerda quantos os indicados pelo 2º argumento. Pelo lado oposto (direita) entram zeros. Exemplo: <pre>mov ax, 0x1234 shl ax, 4</pre> <code>ax</code> tem agora <code>0x2340</code> . |
| <code>shr</code> | <i>Shift right</i> (deslocamento à direita). Semelhante a <code>shl</code> , mas funciona para o lado oposto (direita). Ver <code>shl</code> . Exemplo: <pre>mov ax, 0x1234 shr ax, 4</pre> <code>ax</code> tem agora <code>0x0123</code> . |
| <code>rol</code> | <i>Rotate left</i> (rodar à esquerda). Semelhante a <code>shl</code> , mas em vez de entrarem sempre zeros pelo lado oposto, entram os bits que saem (daí “rodar”). Ver <code>shl</code> . Exemplo: <pre>mov ax, 0x1234</pre> |

```
rol ax, 4
```

ax tem agora 0x2341.

ror *Rotate right* (rodar à direita). Semelhante a **rol**, mas funciona para o lado oposto (direita). Ver **rol**. Exemplo:

```
mov ax, 0x1234
ror ax, 4
```

ax tem agora 0x4123.

Operações de *string*

stosS *Store String* (guardar na *string*). Dependendo de **S**, escreve **al**, **ax** ou **eax** em memória no endereço em **edi** e soma (subtrai se a *flag D* for 1) um valor adequado a **edi**. Ver **lodss** em aula anterior.

Exemplo:

```
section .data
dados    db    1, 2, 3
section .text
        mov  edi, dados
        mov  al, 8
        cld
        stosb
```

O primeiro byte de **dados** terá agora o valor 8, e **edi** terá o endereço do 2.

lodss *Load string, move string e compare string*. Operações alternativas de *strings* explicadas em aulas posteriores. **lodss** já foi explicada em aula anterior.

movss

cmpss

Agora explicar o código torna-se mais fácil.

A secção de dados não tem nada de novo.

Todos os blocos da secção de código são conhecidos, excepto o 3, 4 e 5.

O bloco 4 é trivial: incrementa uma posição de memória. Como esta instrução não tem nenhuma pista sobre que tamanho de dados existe naquela posição de memória (e.g.: um **mov [num]**, **eax** seria de 32 bits porque está a usar um registo de 32 bits) é usada a directiva **dword** antes dos parêntesis rectos para dar a indicação que estamos a trabalhar com uma *double-word* (32 bits).

De facto, a lista de directivas para este propósito é:

| Endereça | |
|--------------------|---------------------|
| <code>byte</code> | <i>Bytes</i> |
| <code>word</code> | <i>Words</i> |
| <code>dword</code> | <i>Double-words</i> |
| <code>qword</code> | <i>Quad-words</i> |

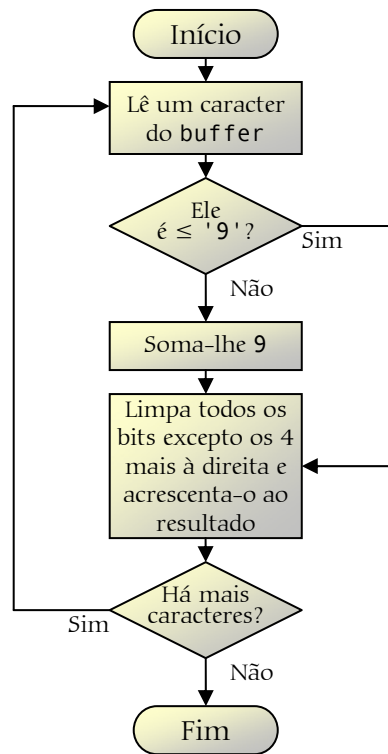
Os blocos 3 e 5 efectuam as conversões de *string* (em hexadecimal) para inteiro e vice-versa, respectivamente.

O algoritmo usado nestas conversões não é específico de Assembly, mas genérico a qualquer linguagem de programação. Também não é a única solução correcta para o problema de conversão.

De forma a funcionar com qualquer tamanho de *string* que se escreva no teclado, a conversão *string* para inteiro converte a primeira caracter a caracter, representando cada um destes um dígito hexadecimal. Assim que a conversão estiver pronta, abrimos espaço num registo que guarda “o resultado até agora” (`edx`) fazendo um deslocamento para a esquerda do valor que lá está, e acrescentamos o novo dígito convertido. Repetimos o processo até se esgotarem caracteres.

A conversão de caracter para dígito é razoavelmente simples, uma vez que a tabela ASCII dispõe os dígitos por ordem numérica e as letras por ordem alfabética. Se eu subtrair `0x30` a um caracter com um dígito numérico, transformei um número entre `0x30` e `0x39` (códigos ASCII dos dígitos numéricos) num número entre 0 e 9. A única dificuldade é que a seguir ao caracter '9' em ASCII não vem a maiúscula 'A', pelo que tenho de fazer um teste comparativo para saber o que estou a converter (um caracter numérico ou uma letra).

Esquemáticamente (explicado na aula):



A conversão inversa usa o mesmo raciocínio, mas de forma inversa. Mais detalhes na aula.

Repare-se que no bloco 5 temos a seguinte linha:

```
add    eax, 'A' - 0xA
```

Aqui, a expressão 'A' - 0xA envolve duas constantes pelo que o resultado pode ser calculado pelo `nasm`, sem gerar novas instruções Assembly. Essa é a única razão pela qual esta expressão é permitida aqui.

Exercícios

Cria um novo programa que:

1. Elimina o bloco 8, mantendo o mesmo resultado. Dica: acrescenta ao final do `buffer` o caracter necessário.
2. Trabalha com números em octal (3 bits por dígito octal) em vez de hexadecimal.

Correcção na aula seguinte.

Aula 07

Correcção do exercício da aula anterior, na aula.

Hoje vamos resolver um novo programa: o `a07-converte2a.asm`. Monta-o* e corre-o. O que faz?.

`a07-converte2a.asm`

Este programa é igual em funcionalidade ao `a06-converte2.asm`, mas o bloco 3 está modificado e tem um novo bloco 10. Ele usa várias novas instruções novas para trabalhar. Elas (e outras da mesma família) são as seguintes:

| Gestão da pilha | |
|--|--|
| <code>push</code> | Coloca um valor de 16 ou 32-bits (que segue a instrução) na pilha. |
| <code>pop</code> | Retira um valor de 16 ou 32-bits da pilha para o registo/memória que segue a instrução. |
| Chamadas a sub-rotinas / funções | |
| <code>call</code> | Chama (<i>calls</i>) uma sub-rotina/função. Guarda na pilha o endereço de retorno. |
| <code>ret</code> | Retorna ao endereço chamador (que deverá estar na pilha), terminando a sub-rotina/função. |
| Endereçamento base+deslocamento | |
| <code>[r]</code> <code>[r₁+r₂]</code> <code>[r*s]</code> <code>[r₁+r₂*s]</code> | Quando usado em qualquer instrução, lê ou escreve na memória no endereço respectivo. <code>r</code> , <code>r₁</code> e <code>r₂</code> são registos e <code>s</code> é um factor de escala (<i>scaling factor</i>) que pode ser 2, 4 ou 8. A qualquer uma destas formas de endereçamento pode ainda ser adicionado ou subtraído um número inteiro (e.g.: <code>[r+i]</code>). Esta é a máxima complexidade que se pode ter nas expressões que se podem usar ao endereçar memória. |

* Se o teu Linux te obriga a montar os programas com `“-f elf64”` então não vais conseguir montar este. Isto porque em modo de 64 bits não podes manipular a pilha com 16 ou 32 bits.

Agora explicar o código torna-se mais fácil.

Aquilo que se fez foi colocar o antigo bloco 3 numa sub-rotina/função independente. Chama-se essa sub-rotina com um `call`, e esta deve terminar com um `ret`, o que a faz retornar ao ponto de onde foi chamada.

Claro que o `ret` só consegue fazer a sua função se tiver “anotado” algures o ponto para onde deve voltar. Isso de facto foi o que a instrução `call` fez, guardando o ponto de retorno em memória, numa estrutura automática chamada de pilha.

Pilha

A pilha do CPU não é mais do que um *buffer* (espaço de trabalho) em memória que é usado numa sequência muito previsível. Ela guarda números de 16 ou 32 bits (nunca 8 bits) que podem ser valores de registos, endereços, ou outros.

Estes valores são guardados e retirados pela mesma ordem que se guarda e retira um livro de uma pilha de livros:



O último livro a ser colocado na pilha (o do topo), será o primeiro livro a ser retirado (*last in, first out*, ou LIFO).

Por razões históricas, as pilhas nos CPUs que estamos a estudar crescem para baixo (para endereços menores), mas o conceito é o mesmo, só que invertido. Temos instruções especiais para colocar e remover explicitamente elementos da pilha (`push` e `pop` descritas acima) e instruções que a usam implicitamente (como `call` e `ret`).

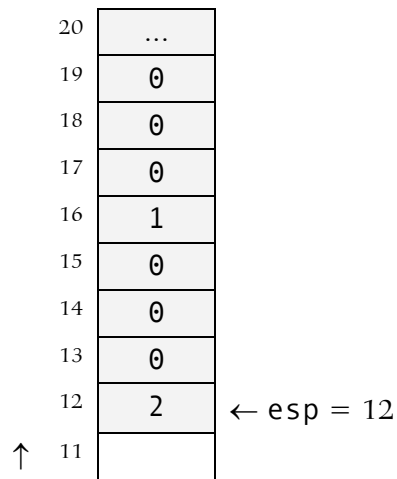
Existe então um registo especial (`esp` – *Stack Pointer*) que mantém o endereço do topo da pilha. Todas as instruções que usam a pilha, escrevem em memória nesse endereço (e usando o segmento de pilha – `ss`), e actualizam `esp` automaticamente.

Exemplo

Após executar estas instruções:

```
mov esp, 20
mov eax, 1
mov ebx, 2
push eax
push ebx
```

A pilha terá o seguinte aspecto:



Exemplo explicado em detalhe na aula.

Questão: porque usam então as instruções `call` e `ret` uma estrutura que tem este comportamento LIFO? Dica: pensa como se deveria comportar o CPU se uma sub-rotina A chama outra sub-rotina B que por sua vez chama outra sub-rotina C.

Passar argumentos a sub-rotinas/funções usando a pilha

Dada a flexibilidade da pilha, é comum ser usada para se passar argumentos a sub-rotinas/funções. Isto é tão comum que temos um registo especial (`ebp` – *Base Pointer*) que sempre que é usado no endereçamento de memória, assume o segmento da pilha. Ele foi desenhado para auxiliar ao acesso a argumentos passados desta forma.

No bloco 3 passamos os argumentos que precisamos fazendo um `push` para cada. Depois da função terminar asseguramo-nos que “retirámos” esses argumentos da pilha com

```
add esp, 8
```

Dentro da sub-rotina `conv2num` no bloco 10, vamos ler esses argumentos. Para tal guardamos temporariamente o `ebp` na própria pilha (para a sub-rotina não o “estragar”), e copiamos o endereço do topo da pilha (`esp`) para `ebp`. A partir deste momento podemos aceder aos argumentos que estão na pilha a partir de `ebp`, e mesmo assim podemos fazer mais `push/pop` e `call/ret` (que usam a pilha) sem nos confundirmos com os novos valores de `esp` nesses instantes.

Acéder aos argumentos é agora uma questão de endereçar a memória com base em `ebp`. Código explicado na aula.

Exercício

Altera o bloco 5 deste programa para também ele ser uma sub-rotina independente, recebendo os mesmos dois argumentos que o antigo bloco 10 (se ambos forem necessários). Correção na aula seguinte.

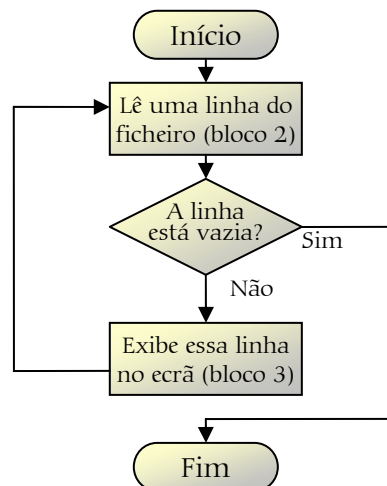
Aula 08

Hoje vamos resolver um novo programa: o `a08-le2.asm`. Monta-o e corre-o. O que faz?

Leitura de ficheiros: `a08-le2.asm`

Este programa lê um ficheiro de texto e exhibe-o no ecrã. Vamos hoje ver como trabalhamos com ficheiros com *Assembly* em Unix/Linux.

O raciocínio central deste programa não é ler o ficheiro todo de uma vez e copiá-lo para o ecrã: isso limitaria o tamanho máximo do ficheiro que podia ser copiado aqueles que coubessem em memória. Em vez disso, o raciocínio é ler uma linha de cada vez e copiá-la para o ecrã, repetindo o processo enquanto houver linhas:



De resto a estrutura deste programa é muito semelhante aos anteriores, não existindo instruções novas. Existem só 3 novos conceitos:

1. Abrir o ficheiro (bloco 1)
2. Quando peço ao Linux para ler uma linha do ficheiro, ele diz-me que a quantidade de bytes lidos é zero se ocorreu um erro ou já não há mais nada para ler (e.g.: cheguei ao fim do ficheiro) (bloco 2)
3. Fechar o ficheiro (bloco 4)

Abrir/fechar ficheiros

Qualquer programa pode precisar de trabalhar com mais do que um ficheiro ao mesmo tempo. Posso precisar de copiar o ficheiro A para o ficheiro B, posso

precisar de cruzar o ficheiro de facturas **F**, com o ficheiro de pagamentos **P** para gerar o ficheiro de avisos de pagamento **A**, etc..

Seria muito lento ter de procurar e localizar o ficheiro em disco (seguindo a estrutura de directórios indicada) e validar as minhas permissões de acesso a cada directório e ao ficheiro correspondente em cada pequena leitura/escrita ao mesmo (e.g.: em cada leitura de uma linha do nosso programa).

Nasce assim o conceito de “abrir” um ficheiro:

Trabalhar com ficheiros:
Abrir → Ler/Escriver → Fechar

“Abrir” um ficheiro serve para o localizar, validar permissões e preparar estruturas do sistema operativo para que as operações de leitura e/ou escrita que se sigam sejam rápidas. O conceito de “fechar” o ficheiro serve para dizer ao sistema operativo para libertar essas estruturas de forma a poupar memória para mais ficheiros.

Handle (“maçaneta” ou número de ficheiro)

Como dizemos nós ao sistema operativo, de qual dos nossos ficheiros abertos queremos ler/escrever? Com um *handle* (“maçaneta”). Este é um número que o sistema operativo nos dá quando abrimos um ficheiro e lhe temos de devolver para nos referirmos ao mesmo mais tarde.



Todas as operações sobre um ficheiro (leitura, escrita e fecho, entre outras) à excepção de abrir, recebem um *handle* para identificar o ficheiro em que operam.

ASCIIZ

Repara como é indicado o nome do ficheiro ao sistema operativo. Para ler/escrever dados precisamos de poder usar qualquer valor de um byte. É por isso que para indicar dados precisamos de dois registos: um para o endereço inicial deles, e outro para o tamanho.

Mas não podemos usar qualquer caracter no nome de um ficheiro (“*” e “?” são inválidos, entre muitos outros). Então usa-se uma “sequência de caracteres terminada no ASCII NUL” (byte com o valor zero) – abreviada ASCIIZ – já que o caracter nulo (NUL) não é válido em nomes de ficheiros ou directórios. Assim apenas precisamos de um registo para indicar o nome do ficheiro ao sistema operativo.

Restantes detalhes explicados na aula.

Vamos a seguir resolver um outro programa: o `a08-escr.asm`. Monta-o e corre-o. O que faz?

Escrita para ficheiros: `a08-escr.asm`

Este programa cria um ficheiro de texto. Todos os blocos são conhecidos, a este ponto.

Note-se que como aqui podemos criar um ficheiro novo, é necessário indicar no registo `edx` quais devem ser as permissões Unix iniciais desse ficheiro, quando o vamos “abrir” (note-se como usamos a mesma função para abrir um ficheiro existente ou para criar um novo ficheiro – que fica aberto). No bloco 1 estamos a indicar as permissões `666` em octal (i.e., leitura e escrita para todos), que serão depois “filtradas” pelo `umask` antes do ficheiro ser criado.

Mais detalhes na aula.

Exercício

Crie um programa que lê dados do teclado e os copia para um ficheiro fixo, semelhante ao `a08-le2.asm`. O programa deve terminar se eu escrever uma linha vazia.

Correcção na aula seguinte.

Aula 09

Correcção do exercício da aula anterior: vamos começar como base com o programa `a08-le2.asm`.

Ele já tem um ciclo que lê uma linha e a escreve (blocos 2 e 3). Mas o bloco 2 deve ler do teclado em vez do ficheiro, por isso altero

```
mov ebx, [fich_handle]
```

para

```
mov ebx, 0
```

Seguindo o mesmo raciocínio, também quero que o bloco 3 escreva para o ficheiro em vez do ecrã e então altero

```
mov ebx, 1
```

para

```
mov ebx, [fich_handle]
```

Claro que isto só funciona se tiver aberto o ficheiro para lhe escrever (ou seja, com um valor adequado na variável `fich_abertura`).

Então na secção de dados, altero o valor dessa variável para `0101q` (o mesmo que está no programa `a08-escr.asm`). Também tenho de adicionar ao topo do bloco 1 a mesma instrução inicial que está ao topo do mesmo bloco nesse programa:

```
mov edx, 0666q
```

Estamos quase prontos. Se o testes o programa funciona, mas não pára quando escreves uma linha vazia. Porquê?

É que as linhas lidas do teclado têm sempre o carácter `0xA` (*Enter*) ao final (ver aula anterior na resolução do exercício do `nome.asm`). Então uma linha “vazia” tem na realidade... 1 carácter!

Assim alteramos no bloco 2, a instrução `cmp` para passar a ser:

```
cmp eax, 1
```

Testa: deve estar a funcionar.

Hoje vamos resolver um novo programa: o `a09-calculadora.asm`. Monta-o e corre-o. O que faz?

Operações aritméticas: `a09-calculadora.asm`

Este programa simula uma calculadora hexadecimal muito simples. Este programa já não vai ser explicado bloco a bloco – será que o consegues entender sozinho?

Algumas instruções novas:

| Operações aritméticas* | |
|------------------------|--|
| <code>mul</code> | <p>Multiplica <code>eax</code> pelo único argumento de 32 bits que a instrução recebe. O resultado é sempre um número de 64 bits cujos 32 bits mais significativos (i.e., “da esquerda”) ficam em <code>edx</code> e os 32 bits menos significativos (i.e., “da direita”) ficam em <code>eax</code>. Isto costuma representar-se como <code>edx:eax</code>.</p> <p>Exemplo:</p> <pre>mov eax, 0xFF123456 mov ebx, 0x100 mul ebx</pre> <p><code>edx:eax</code> tem agora <code>0x000000FF:0x12345600</code>.</p> |
| <code>div</code> | <p>Divide <code>edx:eax</code> (ver <code>mul</code>) pelo único argumento de 32 bits que a instrução recebe. O resultado fica em <code>eax</code> e o resto da divisão em <code>edx</code>.</p> <p>Exemplo:</p> <pre>mov edx, 0x000000FF mov eax, 0x123456AB mov ebx, 0x100 div ebx</pre> <p><code>eax</code> tem agora <code>0xFF123456</code> e <code>edx</code> tem <code>0x000000AB</code>.</p> |
| Salto condicional | |
| <code>jecz</code> | <p>Salta se <code>ecx</code> é zero (<i>jump if ecx is zero</i>). Esta instrução complementa a instrução <code>loop</code> para lidar com ciclos onde <code>ecx</code> pode ser logo à partida <code>0</code>.</p> |

Questão: porque será que a instrução `div` assume o seu argumento da esquerda como um número de 64 bits (`edx:eax`)? Dica: pensa numa multiplicação de um número por uma fracção.

* `mul` e `div` também podem ser usadas com argumentos de 8 e 16 bits. Consulta o manual de referência [IA32-MAN] para mais detalhes (ver Bibliografia).

Detecção de excesso (overflow)

A detecção de quando o resultado não cabe em 32 bits (situação habitualmente chamada de excesso (*overflow*)), pode ser feito da seguinte forma:

- Para a adição e subtração, a *flag carry* tem o valor do 33º bit de resultado (i.e., o “transporte” final). Se este bit for 1, o resultado não cabia em 32 bits!
- No caso da multiplicação detectar *overflow* é trivial: se o resultado fica em `edx:eax...` quando é que isto não cabe em 32 bits?*
- No caso da divisão, se os dois argumentos originais eram de 32 bits, o resultado é sempre \leq ao maior deles, excepto numa situação que temos de testar antes de fazer a divisão. Que situação é essa?†


Detecto a *flag carry* a 1 com o salto `jc`.

Múltipla precisão

As adições e subtracções já estão preparadas para lidar com números inteiros arbitrariamente grandes.

Imagina que quero somar dois números de 64 bits (*quad-words*). Posso dividir cada um em dois números de 32 bits que posso somar com `add`:

| | | | | |
|---|------------------|------------------|------------------|------------------|
| | 0000111100001110 | 0000111100001110 | 1000111100001110 | 0000111100001110 |
| + | 1111000011110000 | 1111000011110000 | 1111000011110000 | 1111000011110000 |
| = | 1111111111111110 | 1111111111111111 | 0111111111111110 | 1111111111111110 |


Carry = 1

Começo a fazer a soma da direita para a esquerda, como sempre. Quando somo os primeiros números de 32 bits, a *flag carry* tem o 33º bit, ou seja o transporte após a última soma do 32º bit de ambos os argumentos. No exemplo acima, a *flag carry* teria 1 pois $1b + 1b = 10b$. Então para “continuar” a somar os 32 bits da esquerda, teria apenas de fazer a soma normalmente e somar também o *carry*!

* Quando após a multiplicação, `edx ≠ 0`! O processador ajuda-nos e coloca a *flag carry* a 1 nesse caso.

† Quando o segundo argumento da divisão é zero! Matematicamente o resultado seria $\pm\infty$ (ou indeterminação no caso $0/0$, mas isso significa que não sabemos se o resultado nesse caso é $+\infty$ ou $-\infty$, pelo que continua a ser uma situação de *overflow*).

Isto consegue-se com uma instrução especial: `adc` (*add with carry*). Para a subtração temos o raciocínio inverso e a instrução `sbb` (*subtract with borrow*):

| Operações de múltipla precisão | |
|--------------------------------|--|
| <code>adc</code> | Somar com o transporte (<i>add with carry</i>). Idêntica a <code>add</code> , mas também soma o valor da <i>flag carry</i> ao resultado. |
| <code>sbb</code> | Subtrair com o transporte (<i>subtract with borrow</i>). Idêntica a <code>sub</code> , mas também subtrai o valor da <i>flag carry</i> do resultado. |

Exemplo:

```
section .data
num1:      ; dq 0x0F0E0F0E8F0E0F0E
num1_l     dd 0x8F0E0F0E
num1_h     dd 0x0F0E0F0E

num2:      ; dq 0xF0F0F0F0F0F0F0
num2_l     dd 0xF0F0F0F0
num2_h     dd 0xF0F0F0F0

resultado: ;dq 0
resultado_l dd 0
resultado_h dd 0

section .text
        mov     eax,          [num1_l]
        add     eax,          [num2_l]
        mov     [resultado_l], eax
        mov     eax,          [num1_h]
        adc     eax,          [num2_h]
        mov     [resultado_h], eax
```

Depois deste código ter corrido, estará em `[resultado]` a soma de 64 bits de `[num1]` com `[num2]`. As instruções `mov` não alteram *flags*, por isso é que posso ter instruções `mov` entre as duas adições. Lembra-te que os números são guardados com os seus bytes do menos significativo para o mais significativo (*little-endian* – ver aula anterior), pelo que os 32 bits “da direita” estão em `[num1_l]=[num1]`, etc., e os 32 bits “da esquerda” estão em `[num1_h]=[num1+4]`, etc.. Se isto te confunde, vê também o exemplo na página seguinte.

As operações de aritmética binária (`and`, `or`, `xor` e `not`), como funcionam com pares de bits e não propagam o resultado para outros bits, só têm de ser repetidas para todos os números de 32 bits que compõem o nosso número de múltipla precisão. Mas o mesmo não se pode dizer de deslocamentos.

Quando desloco um número de múltipla precisão 1 bit para a esquerda (por exemplo), posso começar por fazer `shl` (*shift left*) nos 32 bits mais da direita, mas não posso repetir essa operação, pois quero que o bit que “saiu” pela esquerda destes primeiros 32 bits, “entre” pela direita dos 32 bits seguintes.

Por exemplo:

```
0000111100001110 0000111100001110 | 1000111100001110 0000111100001110
<< shift left
-----
= 000111100001110 0000111100001110 | 1 000111100001110 0000111100001110
                                         ↺
                                         Carry = 1
```

Nos 32 bits da esquerda, entrou um bit com o valor 1 pela direita, pelo que obviamente não posso repetir o `shl`. Mas tenho sorte: quer o `shl` quer todas as outras operações de deslocamentos e rotações colocam na *flag carry* o último bit que foi deslocado “para fora”. Então nos 32 bits seguintes (da esquerda) só precisamos de usar uma instrução que “faça entrar” a *flag carry*, em vez de zero.

Isto consegue-se com uma instrução especial, `rcl`:

| Operações de múltipla precisão | |
|--------------------------------|--|
| <code>rcl</code> | <p>Rodar à esquerda através da <i>flag carry</i> (<i>rotate through carry, left</i>). Idêntica a <code>rol</code>, mas onde o argumento a ser rodado tem mais um bit (a <i>flag carry</i>).</p> <p>Por exemplo,</p> <pre>mov ax, 0 stc ; set carry flag rcl ax, 1</pre> <p>O bit que entra pela direita em <code>ax</code> vem da <i>flag carry</i>, e o bit mais à esquerda de <code>ax</code> e que vai sair, vai para esta mesma <i>flag</i> também. Assim após estas instruções, <code>ax</code> tem 1 e <i>carry</i> tem zero.</p> |
| <code>rcr</code> | <p>Rodar à direita através da <i>flag carry</i> (<i>rotate through carry, right</i>). Idêntica a <code>rcl</code>, mas funciona no sentido oposto. Ver <code>rcl</code>, acima.</p> |

Exemplo:

```
section .data
num1      dd 0x8F0E0F0E, 0x0F0E0F0E
;         dq 0x0F0E0F0E8F0E0F0E

section .text
        shl dword[num1], 1
        rcl dword[num1+4], 1
```

Depois deste código ter corrido, `[num1]` estará deslocado à esquerda por 1 bit. Não te esqueças que `rcl` e `rcr` só funcionam bem para este propósito se estiveres a deslocar um único bit.

Neste exemplo usámos uma única variável `num1` com duas *double-words*. Novamente, a sua parte “da direita” (*low*) é `[num1]` e a sua parte “da esquerda” (*high*) é `[num1+4]`.

Apesar de não existirem instruções de multiplicação e divisão de múltipla precisão, podemos usar as adições e deslocamentos de múltipla precisão em algoritmos de multiplicação e divisão.

Exercícios

Estes exercícios são muito semelhantes ao que te poderá ser pedido no projecto final da cadeira, pelo que não será feita correcção:

1. Altera a calculadora para fazer detecção de *overflow*.
2. Altera a calculadora para lidar com números de 64 bits.

Em caso de dúvida consulta o professor.

Aula 10

Hoje vamos falar de aritmética inteira com sinal, em complemento para dois. Esta aula pode ser adiada pois precisas de conceitos importantes sobre representação de números inteiros com sinal das aulas teóricas. Precisas de saber o que é o “complemento para dois” e a sua “representação normalizada em binário”.

Operações em inteiros com sinal

Todas as operações aritméticas mencionadas até aqui trabalham sobre números inteiros sem sinal (i.e., positivos e o zero).

Há várias formas de representar números inteiros com sinal. A forma preferida pelas instruções deste CPU em números inteiros é o complemento para dois. A razão para isso é simples: somamos (e subtraímos) números em complemento para dois exactamente da mesma forma que número sem sinal! O resultado é sempre correcto desde que seja interpretado como também ele um número com sinal. Então as instruções para soma e subtracção de números com sinal também são as mesmas: `add` e `sub` (assim como também são as mesmas `inc` e `dec`). Mais detalhes nas aulas (teóricas).

Mas o mesmo não se pode dizer da multiplicação e divisão. Para essas operações temos instruções novas:

| Operações aritméticas, com sinal | |
|----------------------------------|---|
| <code>imul</code> | Multiplicação. Idêntica em tudo a <code>mul</code> , mas assume que todos os argumentos (e o resultado) são números com sinal em complemento para dois. |
| <code>idiv</code> | Divisão. Idêntica em tudo a <code>div</code> , mas assume que todos os argumentos (e o resultado) são números com sinal em complemento para dois. |
| <code>neg</code> | Negação aritmética. Troca o sinal do número que está no registo ou memória do argumento que recebe, assumindo-o como complemento para 2. Exemplo: <pre>mov ax, 0xFFFF neg ax</pre> <code>ax</code> tem agora <code>0x0001</code> . |

A instrução `imul` em particular pode ser usada com dois ou três argumentos em situações particulares que não vamos estudar. Consulta o manual de referência [IA32-MAN] para mais detalhes (ver Bibliografia).

Detecção de excesso (overflow)

As somas e subtracções em números com sinal podem gerar *carry* sem haver *overflow*. Por exemplo, se eu tiver o número 2 e lhe subtrair 5, o resultado é -3 mas o processo de subtracção gera *carry*.

O excesso em números com sinal não é detectado com *carry*, mas no transporte do penúltimo bit para o último, combinado com os sinais (último bit) dos argumentos.

Felizmente temos uma *flag* que já calcula isto e nos diz se houve excesso nestas operações ou não. Chama-se *flag overflow* e tem a abreviatura `O` (letra `O`, não o número zero).

A instrução que nos permite detectar o excesso com adições, subtracções e multiplicações é assim a `jo` (*jump if overflow*), correspondente à `jc` das mesmas operações sem sinal. A forma de detectar excesso na divisão é idêntica à aritmética sem sinal (porquê?). Existe apenas uma excepção, uma única outra divisão que em números com sinal pode resultar em excesso. Consegues descobrir qual é?*

Múltipla precisão

Como as operações de adição e subtracção em números com sinal são as mesmas das respectivas operações em números sem sinal, o mesmo se aplica às instruções usadas para múltipla precisão: `adc` e `sbb`.

Exercício

Esta aula é curta para te dar mais oportunidade de trabalhar nos exercícios da aula anterior. Quando os terminares, adiciona à calculadora da aula anterior a capacidade de trabalhar com números com sinal. Lembra-te de corrigir a detecção de *overflow*.

Não te esqueças que quando corres a calculadora, introduzes um número negativo em complemento para 2. Como introduzes então -1?[†]

* A divisão do número mais próximo de $-\infty$ por -1. Repara: qual é a gama de números disponível em complemento para 2, em 32 bits?

† A calculadora usa números de 32 bits. Então o complemento para dois de 1 (para o transformar em -1) é: `0xFFFFFFFF`.

Aula 11

Hoje vamos falar de aritmética não-inteira (i.e., fraccionária, ou de “vírgula flutuante”). Esta aula pode ser adiada pois precisas de conceitos importantes sobre representação de números de vírgula flutuante das aulas teóricas. Precisas de saber o que é o “significando” (também conhecido como coeficiente ou *mantissa*) e o “expoente” e a sua representação normalizada em binário (formato IEEE 754).

Operações em números fraccionários

Todas as operações aritméticas mencionadas até aqui trabalham sobre números inteiros com ou sem sinal.

Os circuitos do CPU que trabalham com números de vírgula flutuante eram opcionais e comprados à parte até ao 80386. Era um co-processor chamado 80387. Também por esta razão ainda se dá de forma muito notória um nome especial a estes circuitos: a FPU (*floating-point unit* ou unidade de vírgula flutuante).

As instruções da FPU são novas e trabalham com **oito novos registos que guardam um número de vírgula flutuante** de 80 bits e **funcionam como uma pilha**. Por razões de eficiência esta pilha é circular, mas é acedida relativamente ao seu “topo” actual. Por exemplo a notação $ST(0)$ ou $ST0$ significa “*stack top + 0*”.

Todas as instruções da FPU que fazem operações aritméticas lêem os seus argumentos da pilha. Então, para fazer um cálculo, tenho de colocar os valores do mesmo na pilha (com instruções específicas), e só depois executo a instrução aritmética em si. Esta instrução retira da pilha os seus argumentos e substitui-os pelo resultado da operação em si.

Por exemplo, para fazer a conta

$$(5,6 \times 2,4) + (3,8 \times 10,3)$$

tenho de fazer os seguintes passos:

1. Colocar 5,6 na pilha
2. Colocar 2,4 na pilha
3. Executar o produto (na pilha saem os dois valores e fica o resultado)
4. Colocar 3,8 na pilha
5. Colocar 10,3 na pilha
6. Executar o produto (na pilha saem os dois valores e fica o resultado) (não esquecer que o resultado do produto anterior ainda lá está)

7. Executar a soma

Para simplificar estes passos, as instruções permitem geralmente que o último argumento faça parte da instrução em si.

Assim, a sequência final de passos será:

1. Colocar 5,6 na pilha
2. Multiplicar por 2,4 (na pilha sai o 5,6 e fica o resultado)
3. Colocar 3,8 na pilha
4. Multiplicar por 10,3 (na pilha sai o 3,8 e fica o resultado)
(não esquecer que o resultado do produto anterior ainda lá está)
5. Somar o topo da pilha pelo topo da pilha+1

Nota que todos os números de vírgula flutuante usados devem estar em memória e eu devo referir-me a eles pelo seu endereço.

O código pode então ser o seguinte:

```
section .data
valor1      dq  5.6
valor2      dq  2.4
valor3      dq  3.8
valor4      dq 10.3

section .text
                fld  qword[valor1] ; Passo 1
                fmul qword[valor2] ; Passo 2
                fld  qword[valor3] ; Passo 3
                fmul qword[valor4] ; Passo 4
                fadd  st1           ; Passo 5
```

Todas as instruções de vírgula flutuante começam com a letra **f** que é a abreviatura de “*floating-point*”.

As instruções novas são as seguintes:

| Operações de vírgula flutuante | |
|--------------------------------|--|
| fld <i>r</i> | Carregar constante de vírgula flutuante (<i>floating-point load</i>). Faz o equivalente a um push , mas para a pilha de vírgula flutuante, de uma constante guardada numa <i>quad-word</i> em <i>r</i> . |
| fadd <i>r</i> | Soma o topo da pilha (ST0) com <i>r</i> . Substitui o topo da pilha pelo resultado. |
| fsub <i>r</i> | Subtrai do topo da pilha (ST0), <i>r</i> . Substitui o topo da pilha pelo resultado. |
| fmul <i>r</i> | Multiplica o topo da pilha (ST0) por <i>r</i> . Substitui o topo da pilha pelo resultado. |

| | |
|---------------------|---|
| <code>fdiv r</code> | Divide o topo da pilha (<code>ST0</code>) por <code>r</code> . Substitui o topo da pilha pelo resultado. |
| <code>fst r</code> | Guardar constante de vírgula flutuante (<i>floating-point store</i>). Faz o equivalente a um <code>pop</code> , mas da pilha de vírgula flutuante, para uma variável <code>r</code> . |

Em qualquer uma destas instruções, `r` pode ser uma referência à memória (e.g.: `qword[numero]`) ou à pilha de vírgula flutuante (e.g.: `ST0`).

Existem inúmeras mais instruções disponíveis, para fazer todo o tipo de operações que encontrarias numa calculadora científica (senos, logaritmos, raízes, arredondamentos, etc.). Consulta o manual de referência [IA32-MAN] para mais detalhes (ver Bibliografia).

Exercício

Pega na calculadora de 64 bits que construístes. Assume que os números que a pessoa introduz são constantes de vírgula flutuante de 64 bits, no seu formato IEEE 754. Assume também que o número hexadecimal que a calculadora vai exibir, é o resultado da operação no mesmo formato IEEE 754.

Altera então as quatro operações da calculadora para fazerem esses cálculos em vírgula flutuante.

Exercícios

As aulas seguintes são de revisão e aprofundamento desta matéria e não estão detalhadas neste guião. Elas servem para te acompanhar na execução do projecto final da cadeira e para cimentar os conhecimentos aqui adquiridos.

Não deixes de ir às aulas.

VirtualBox Linux

O teu professor das aulas práticas pode fornecer-te uma máquina virtual Linux (Xubuntu para ser mais específico) preparada para correr no Sun VirtualBox.

Isto serve apenas para te ajudar a mais rapidamente começares a trabalhar e não vai servir para outras cadeiras nem é um exemplo de um Linux moderno ou popular.

Instalar o software base

Esta máquina não está disponível pela Internet pois é muito grande (quase 850Mb), pelo que terás de levar o teu computador portátil para a escola para a receberes.

Os ficheiros que o professor te irá dar são 3:

- Sun VirtualBox 3.0.10
- 7-Zip 4.57
- Imagem do disco Xubuntu 8.10 comprimida com 7-Zip*

Os dois primeiros ficheiros são executáveis, mas foram gravados com a extensão “e_e” para limitar propagação de vírus antes de te chegarem às mãos. Se mesmo assim estiveres preocupado com vírus, podes descarregar esses dois programas:

- Sun VirtualBox em: <http://www.virtualbox.org/>
- 7-Zip em: <http://www.7-zip.org/>

Ambos são gratuitos.

Se confiares nesses ficheiros, tens de trocar a extensão (terminação do nome) de ambos para “exe”. A seguir faz duplo clique em cada um e segue os passos exibidos no ecrã para os instalar.

Instalar a imagem do Xubuntu

O teu professor escolheu o Xubuntu para ser o sistema Linux que te oferece já configurado por este ser muito pequeno e fácil de usar ($\pm 2,5$ Gb após descompressão da imagem).

* Sim, é verdade que se fosse comprimida em Zip normal não seria necessário instalar qualquer programa adicional para a descomprimir, mas em Zip normal a imagem não cabe numa memória USB de 1Gb...

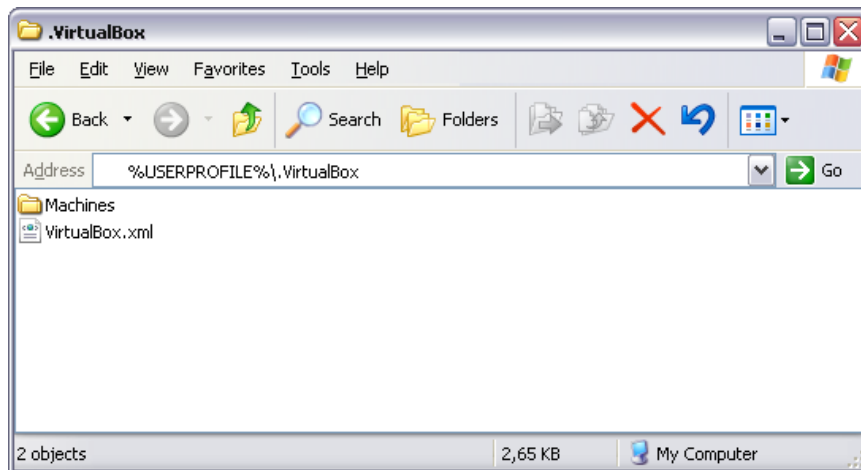
O Xubuntu não é mais do que um popular sistema Linux (Ubuntu) com o ambiente de trabalho Xfce (rápido e pequeno). Podes saber mais sobre ele em:

<http://www.xubuntu.org/>

Para extrair os ficheiros desta imagem, deves abrir a pasta principal do VirtualBox. Para isso, abre uma pasta qualquer no teu computador, e escreve na barra de endereços:

`%USERPROFILE%\VirtualBox`

(maiúsculas e minúsculas não são importantes, mas coloca todos os “%”, “\” e “.” sem espaços tal como exibidos acima). Carrega em *Enter*. Deves ficar com algo semelhante a isto:









É normal que a barra de endereços mude para outra coisa no teu computador.

Se na pasta que abriste não existe uma pasta com o nome “HardDisks” (tal como no exemplo acima), tens de a criar.

Entra nessa pasta (HardDisks) e extrai o ficheiro “.7z”, arrastando-o com o botão da direita do rato para lá, largando-o e escolhendo do menu que surge: 7-Zip → Extract Here.

Se ficares com estes ficheiros na pasta, tudo correu bem!

-  Ubuntu.vmdk
-  Ubuntu-s001.vmdk
-  Ubuntu-s002.vmdk
-  Ubuntu-s003.vmdk
-  Ubuntu-s004.vmdk
-  Ubuntu-s005.vmdk

Podes apagar o ficheiro “.7z”.

Criar a máquina virtual do Xubuntu

Arranca o Sun VirtualBox a partir do menu de arranque (*Start*). Clica em “Novo” para criar uma nova máquina virtual.

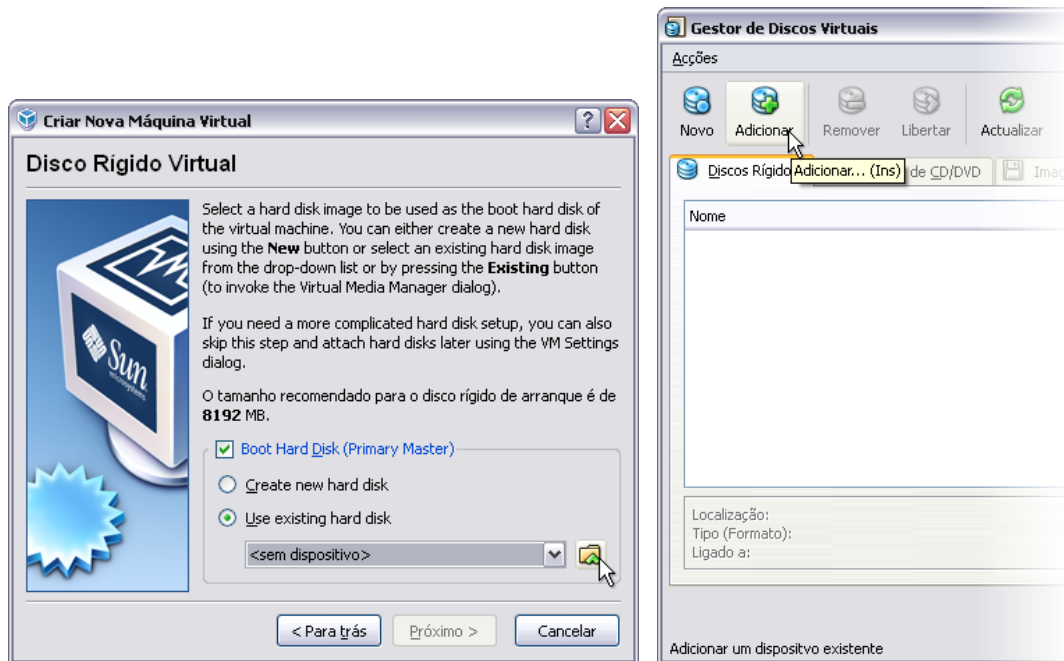


Escolhe os seguintes valores e respostas para as caixas de diálogo que se seguem (Sistema operativo “Linux”, versão “Ubuntu” e 512Mb de RAM).

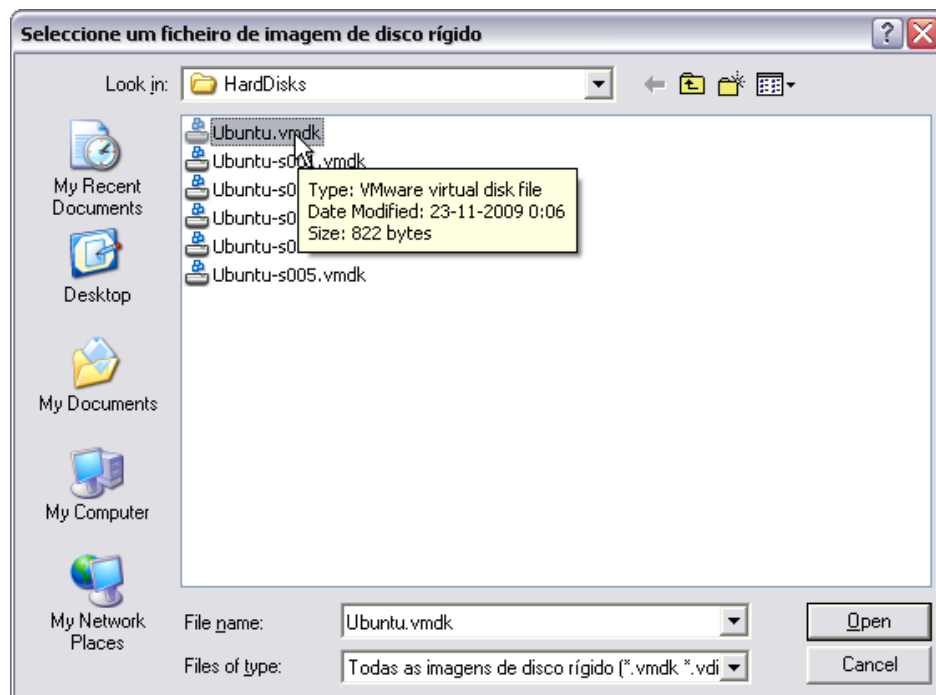


Se o teu computador tiver 1Gb de RAM ou menos, mantém o uso de RAM nos 384Mb recomendados pela Sun VirtualBox, senão o teu computador torna-se muito lento.

Ao escolher o disco rígido virtual, deves localizar os ficheiros que extraíste anteriormente. Escolhe “Usar um disco rígido existente” e clica no símbolo de pasta que está à direita.

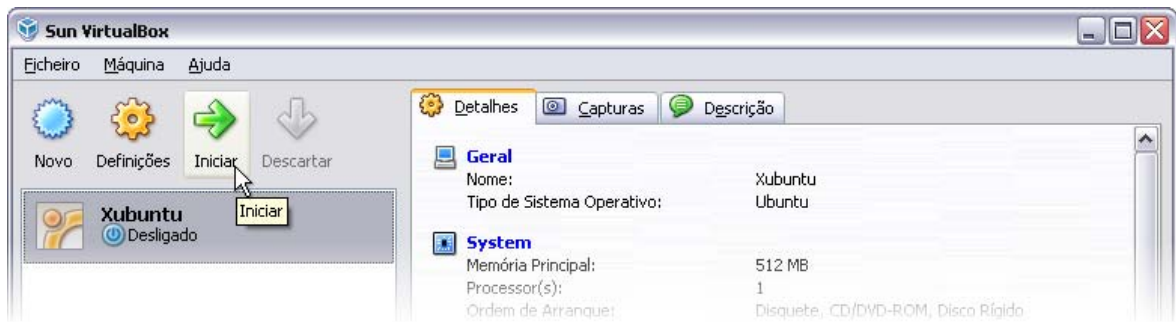


No gestor de discos virtuais (acima), clica em “Adicionar”, escolhe o ficheiro “Ubuntu.vmdk” (abaixo) e acaba a criação da máquina virtual.



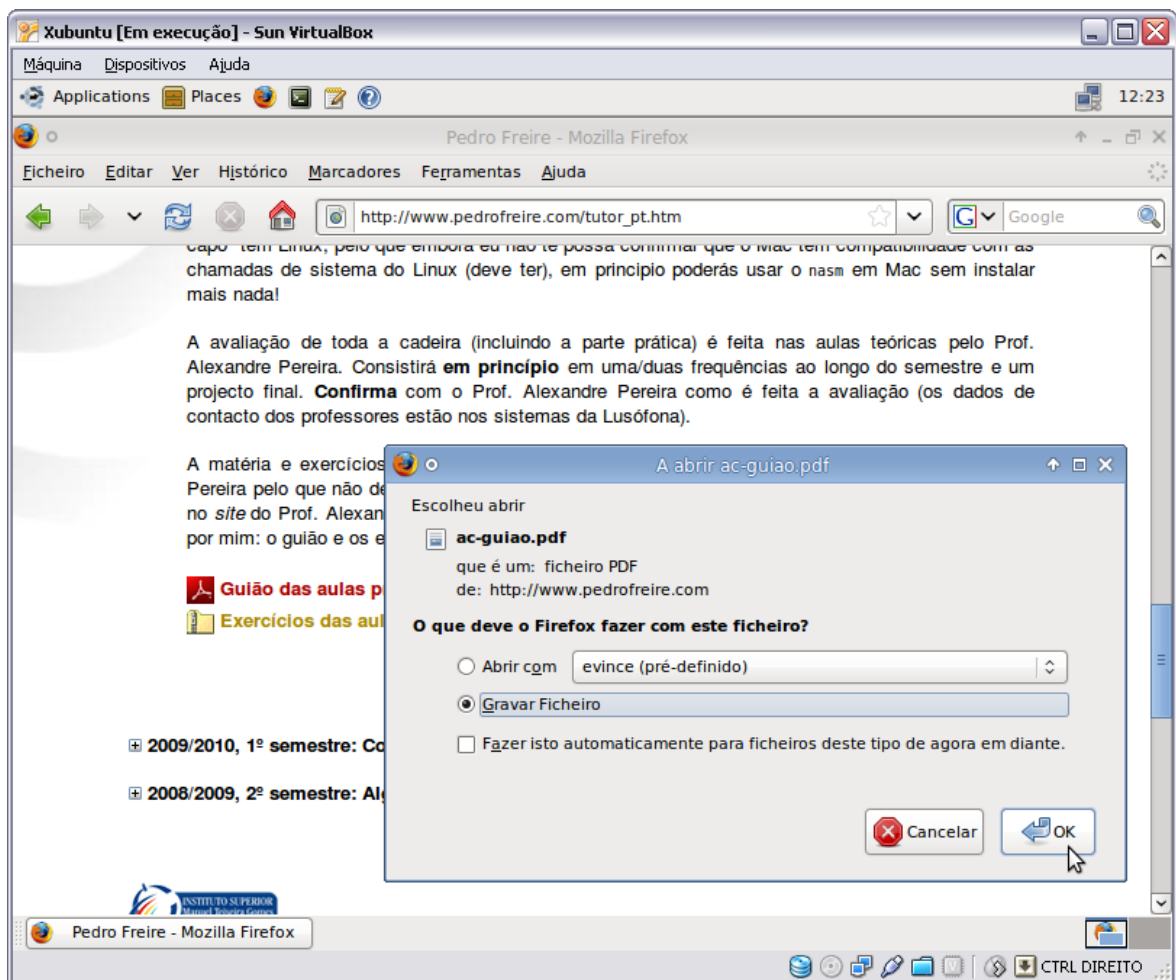
Se não vires estes ficheiros, instalaste incorrectamente a imagem do Xubuntu. Volta à secção anterior e tenta repetir os passos.

A máquina virtual está agora instalada e pronta a arrancar (botão “Iniciar”)!



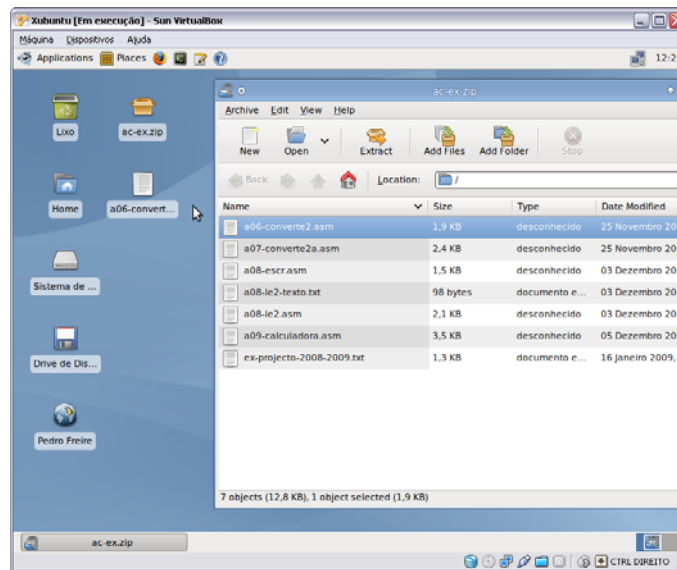
Usar o Xubuntu

Arranca a máquina virtual. Ela tem os programas que mais vais usar logo no painel superior. Experimenta arrancar o Firefox, vai ao *site* do professor e clica no guião e exercícios da aula.

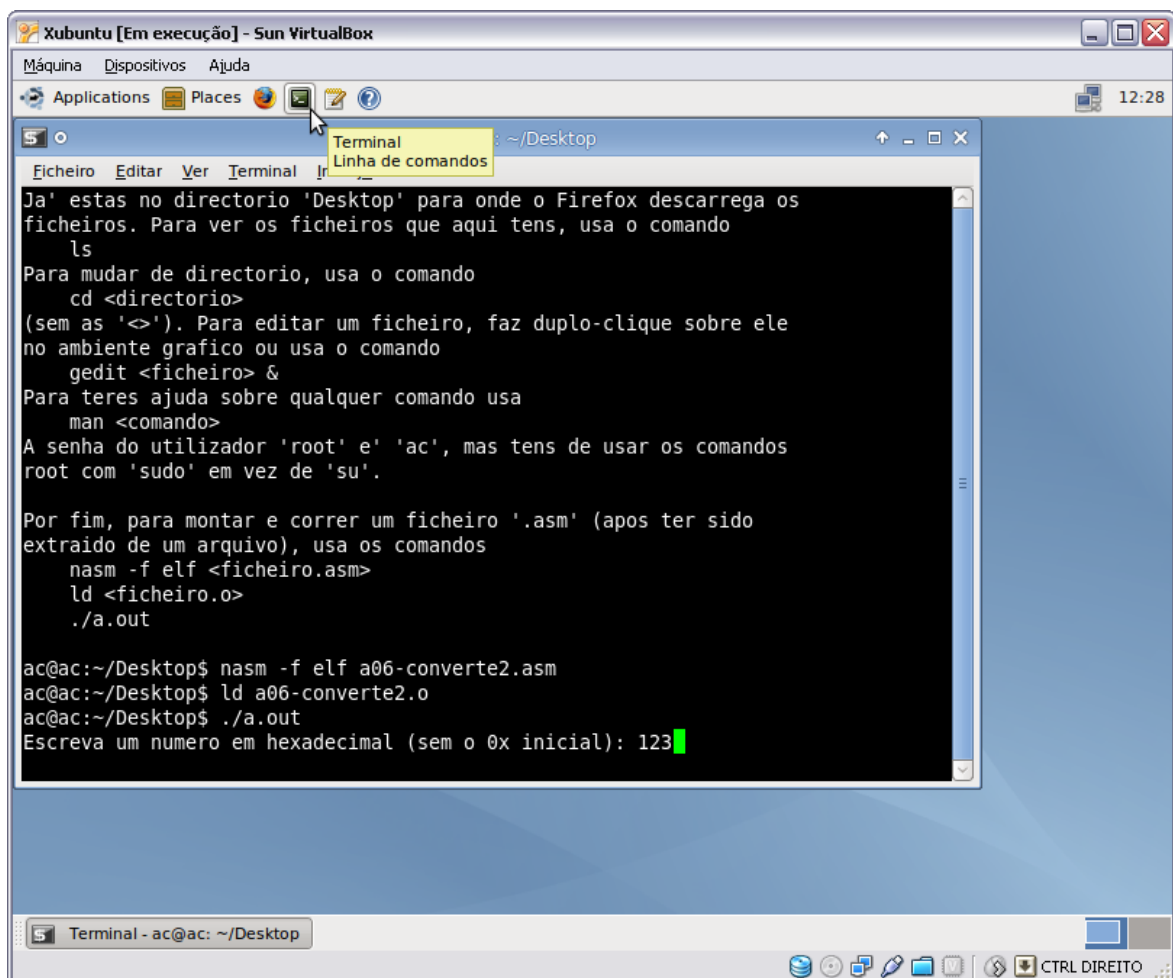


Quando o Firefox te pedir, escolhe gravar o ficheiro. Ele irá gravar tudo o que recebe da Internet no ambiente de trabalho (*desktop*) para o encontrares mais facilmente.

Para poderes montar os programas Assembly das aulas, tens sempre de os extrair dos arquivos. Para isso abre o arquivo que descarregaste com um duplo-clique e arrasta o nome do ficheiro que pretendes de volta para o *desktop*.



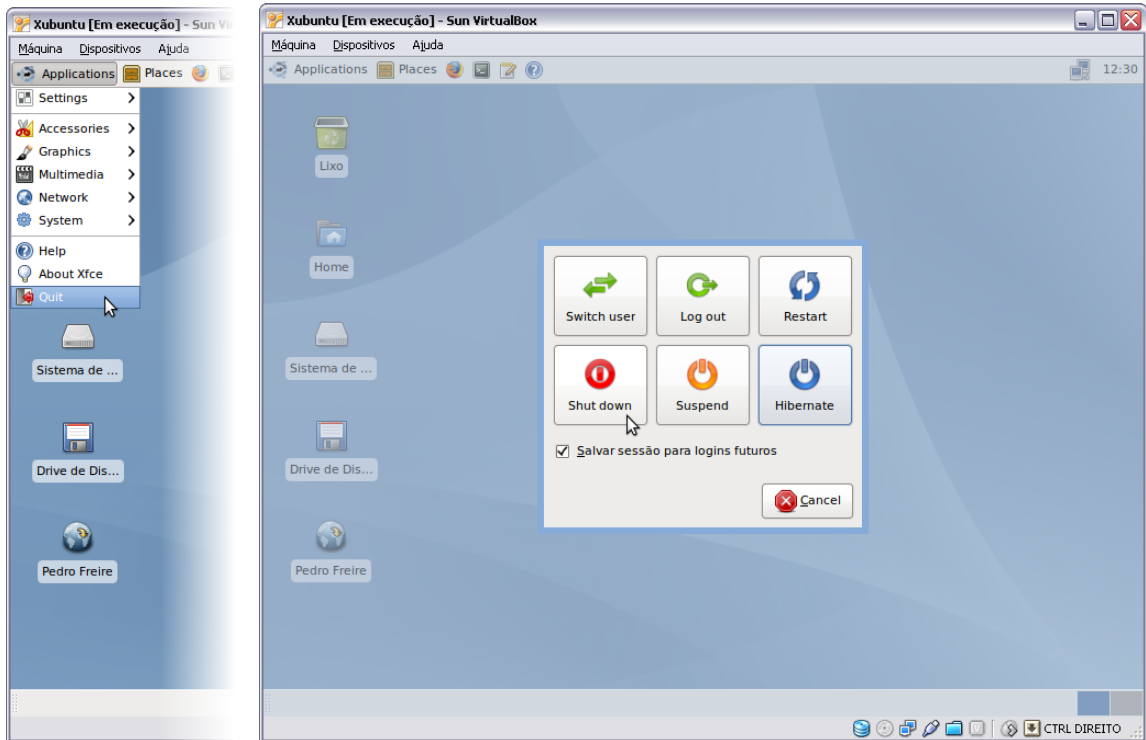
Agora, com os ficheiros extraídos, já podes montá-los. Abre a consola/terminal e corre os comandos referidos na Aula 01.



E é tudo!

Segue este Guião a partir da Aula 01 e aparece nas aulas para tirares dúvidas. Se estragares algo, volta a instalar a imagem do Xubuntu.

Para saíres do Linux, escolhe "Quit" no menu "Applications" do painel superior e depois escolhe "Shutdown". Podes também no menu "Máquina" da Sun VirtualBox escolher "Fechar" e depois "Gravar o estado".



Bibliografia

[IA32-MAN]

"Intel® 64 and IA-32 Architectures Software Developer's Manuals", Intel.

No site da Intel em:

<http://developer.intel.com/products/processor/manuals/>

[80386-PG]

Lance Leventhal,

"80386 Programming Guide", Bantam Computer Books.

Na Amazon do Reino Unido, em:

<http://www.amazon.co.uk/80386-Programming-Guide-PC-Library/dp/055334529X/>

(mais em breve)