NOVA SCHOOL OF SCIENCE & TECHNOLOGY

Arquitetura de Computadores 2022/23

Ficha 4

Tópicos: *Introdução à linguagem assembly (x86_64)* e aos utilitários *assembler, linker (ligador) e debugger (depurador).*

Parte I – Assembly "puro", linker e debugger

Observações: *O objetivo desta secção é utilizar um programa fonte, já fornecido, em assembly, usar o assembler e o linker para produzir o executável, e executá-lo sob controle do debugger.*

1. Considere o seguinte exemplo de programa em *assembly* Intel (o ficheiro com este código está disponível no CLIP com o nome **`hello.s`**):

```
EXIT  = 60                              # usando simbolos para constantes
WRITE = 1


.data                                   # seccao de dados (variaveis)
 msg:     .ascii  "Hello, world!\n"     # um vetor de caracteres
 msglen = (. - msg)                     # msglen representa o tamanho do vetor

.text                                   # seccao de codigo
.global  _start                 # exportar o simbolo _start (inicio do programa)

_start:  mov    $msglen,%rdx      # comprimento da mensagem
         mov    $msg,%rsi         # endereço da mensagem
         mov    $1,%rdi           # escreve no stdout
         mov   $WRITE,%rax       # pedir write ao sistema
         syscall   # chama o sistema

         movq    $0,%rdi
         movq    $EXIT,%rax        # pedir o exit ao sistema
         syscall   # chama o sistema
```

a) Obtenha o executável respetivo e teste a sua execução. Para tal "assemble" o programa fonte e ligue-o para obter o executável, usando a seguinte sequência de comandos:

```
as -o hello.o hello.s
ld -o hello hello.o
```

Execute depois o programa com: **`./hello`**.

b) Execute em seguida o programa **`hello`** sob o controlo do *debugger*, executando-o passo-a-passo. Repare que, à semelhança do que acontecia com o compilador de C, para o *debugger* mostrar o código fonte tal como o escreveu, deve usar a opção **`-g`** no **`as`**. Exemplo:

```
as -g -o hello.o hello.s
ld -o hello hello.o
```

Execute depois o programa com: **`gdbtui hello`** (poderá, se tiver outro *debugger* mais "gráfico", usar esse 😊)

Coloque um *breakpoint* (por exemplo logo em **`_start`**) e use o comando "`layout regs`" para que seja mostrado o estado dos registos do CPU enquanto executa o programa passo-a-passo. Confirme o efeito de cada **`movl`** nos registos do CPU. Pode imprimir o valor das variáveis com o comando *print*, como antes, mas podemos ter de forçar a interpretação correta. Exemplo: "`print (char[14])msg`".

c) Altere agora a mensagem a afixar no ecrã para uma à sua escolha. Execute esta nova versão.

d) Observe agora o código presente no executável usando o comando **`objdump`**. Se o seu programa se chamar "**`hello`**" use o seguinte comando:

`objdump -d hello`

Procure na listagem produzida o código a seguir ao símbolo **`_start`**. Deve verificar que aparecem endereços, a representação hexadecimal do código máquina e respetivas mnemónicas em *assembly*, que devem ser idênticas ao código fonte *assembly*.

e) Repita a experiência de *disassembly* da alínea anterior, mas com um outro executável qualquer, por exemplo um programa seu de uma aula passada (desenvolvido na linguagem C). Procure a função **`main`** e verifique o respetivo código produzido pelo compilador. Note que existe mais código, colocado pelo compilador e *linker*, necessário para o seu programa iniciar a execução pela função **`main`** com os respetivos argumentos, assim como para a chamada das funções da biblioteca do C

2. Parte II – Assembly: vetores

Observações: *O objetivo desta secção é implementar programas em assembly que manipulam vetores.*

2. Complete o programa em *assembly* Intel 64 bits (para Linux), **`vetor.s`**, que calcula o número de elementos iguais a **`x`** existentes no vetor indicado. Use como ponto de partida as seguintes variáveis:

```
.data
vetor: .int  -1, 5, 1, 1, 4  # um vetor de inteiros, experimente com
                              # diferentes tamanhos e com outros valores
len = (. - vetor) / 4         # porque?  O que faz isto?
x:     .int 1                 # elemento a pesquisar
total: .int 0                 # total de valores =s a x encontrados no 'vetor'
```

Note que para resolver este problema terá que utilizar endereçamento indireto. Use o *debugger* para executar o programa passo-a-passo e confirmar que o valor correto é guardado tanto no registo **rax** (tente usá-lo como contador) como na variável **total**.
Se terminar o seu programa assim,

```
mov    %rax,%rdi
mov    $EXIT,%rax       # pedir o exit ao sistema
syscall   # chama o sistema
```

verá que, após a execução terminar e voltar ao *prompt* da *shell*, consegue obter o resultado executando o comando echo $? (Nota: o comando echo tem de ser executado imediatamente depois do programa que trata o vetor terminar; se executar algum outro comando no mesmo "terminal" antes de executar o comando echo, já não obtém a resposta desejada). Por exemplo, com os dados acima, deverá ter algo como

```
$ echo $?
2
```

3. Mantendo apenas os "dados" **vetor** e **len** (já usados no programa anterior), e usando o mesmo tipo de endereçamento (i.e., indireto), calcule agora a soma dos elementos do vetor. Deve guardar o resultado no registo **eax** e o programa deve terminar com as mesmas três linhas de finalização acima indicadas, de forma a que o resultado obtido seja "passado" à *shell* (de forma a poder ser observado com **echo $?**). Complete o "esqueleto" de programa que lhe é fornecido, **somaVetor.s**.

4. Altere o programa `hello.s` acima apresentado para que inclua um passo de pré-processamento antes de imprimir a mensagem. Este pré-processamento deverá percorrer o vetor de carateres e, caso o carácter seja uma letra minúscula (e só neste caso), deverá converter o carácter numa letra maiúscula. Só após realizar o pré-processamento da mensagem é que esta deverá ser impressa no ecrã. Execute e verifique se a mensagem que aparece no terminal tem todos os caracteres em maiúscula. Se necessário recorra ao *debugger* para verificar o seu funcionamento.

5. Escreva um programa em *assembly* Intel para Linux 64bits que faz a soma dos elementos de mesmo índice de dois vetores *vetor1* e *vetor2* e deixa o resultado das somas no *vetor2*. Use como ponto de partida as seguintes variáveis:

```
.data
  vetor1:       .int  -1,  5, 1,  1, 4    # operando 1
  vetor2:       .int   1, -3, 1, -5, 4    # operando 2 e destino
LEN = (.-vetor2)/4
```

Use o *debugger* para executar o programa passo-a-passo e confirmar que calcula os valores corretos. Resolva este exercício utilizando endereçamento *indireto por registo*.

3. Parte II – Assembly: subrotinas

Observações: *O objetivo desta secção é implementar programas em assembly que invocam subrotinas.*

6. Escreva um programa em *assembly* Intel para Linux 32bits que implementa uma subrotina que recebe dois números como argumentos e devolve/retorna o maior dos dois. Resolva este exercício assumindo que os dois argumentos são passados nos registos `%rdi` e `%rsi` e valor de retorno é devolvido no registo `%rax`

7. Escreva um programa em *assembly* Intel para Linux 64bits que implementa uma subrotina para comparar os caracteres de duas cadeias, `cad1` e `cad2`, terminadas com o código zero (*strings* C). A comparação será da "esquerda para a direita" e considera-se uma cadeia A maior que uma cadeia B se o primeiro caráter diferente tiver em A um código ASCII maior que o de B, ou se a cadeia B terminou. O resultado deve ser colocado no inteiro `res` em memória. Caso as cadeias sejam iguais, o resultado é zero; caso a primeira cadeia seja maior, o resultado é 1; caso a segunda seja maior que a primeira, o resultado é -1. Use como ponto de partida os exemplos seguintes:

```
.data
cad1:    .ascii   "hellu\n\0"
cad2:    .ascii   "hello\n\0"
res:           .int      0            # o resultado vai ser 1

.data
cad1:    .ascii   "flor\0"
 cad2:   .ascii   "floresta\0"
res:           .int      0            # o resultado vai ser -1
```