

Arquitectura de computadores 2010/11

Aula prática nº 7 – semana de 11 de Abril de 2011

Tema: Construção de programas em assembly do Pentium com utilização de subrotinas. Utilização de uma biblioteca de subrotinas para fazer chamadas ao sistema

A. Alguns programas usando subrotinas

Manipulação da pilha no Pentium

O Pentium tem uma pilha que segue a disciplina de inserção “Last In First Out”. A pilha é gerida usando um registo do CPU dedicado chamado ESP. O ESP aponta para a última posição ocupada na pilha. Vamos supor que são sempre empilhados 4 bytes (32 bits). As principais operações são *PUSH* e *POP* :

Push_valor

O valor pode ser uma constante, o conteúdo de um registo ou o conteúdo de uma posição de memória.

$$SP = SP - 4$$

$$\text{mem}[SP] = \text{valor}_{7-0} ; \text{bits } 7 \text{ a } 0 \text{ do valor a empilhar}$$

$$\text{mem}[SP+1] = \text{valor}_{15-8}$$

$$\text{mem}[SP+2] = \text{valor}_{23-16}$$

$$\text{mem}[SP+3] = \text{valor}_{31-24}$$

Pop_destino

O destino pode ser um registo ou uma posição de memória

$$\text{destino}_{7-0} = \text{mem}[SP] ; \text{bits } 7 \text{ a } 0 \text{ do destino}$$

$$\text{destino}_{15-8} = \text{mem}[SP+1]$$

$$\text{destino}_{23-16} = \text{mem}[SP+2]$$

$$\text{destino}_{31-24} = \text{mem}[SP+3]$$

$$SP = SP + 4$$

Subrotinas

Chamada de subrotina: instrução CALL etiqueta

Esta instrução faz push do EIP e escreve o valor associado à etiqueta no EIP

Retorno de subrotina: instrução RET

Esta instrução é o equivalente a um pop EIP, isto é, espera-se que o valor que foi empilhado pela instrução CALL seja desempilhado do stack e carregado no EIP. Assim a execução continua na instrução máquina a seguir à instrução CALL que invocou a subrotina corrente

Passagem de parâmetros

Os parâmetros podem ser passados por registos ou empilhados no stack usando a instrução PUSH.

Retorno de resultados

Se há só um valor a retornar é usado o registo eax para esse efeito

Passagem de parâmetros por registos

Podem ser usados os registos EAX, EBX, ECX e EDX para passar parâmetros a uma subrotina. No exemplo seguinte ilustra a subrotina `my_add1` soma dois valores que recebe nos registos EAX e EBX, retornando o resultado em EAX:

```
section .data
val:      dd 0
res:      dd 0
...
section .text
...
    mov     eax, [val]           ; res = my_add( val, 2)
    mov     ebx, 2
    call    my_add1
...                               ; eax tem o valor retornado
    mov     [res], eax
...
my_add1:
    add     eax, ebx
    ret
```

Passagem de parâmetros pela pilha

Se os parâmetros foram empilhados, pode-se usar endereçamento baseado (usando o ESP como registo base) para fazer acesso ao seu valor. O último parâmetro empilhado tem o endereço `[esp+4]`, o penúltimo `[esp+8]`, etc. Na próxima veremos que o registo EBP pode ser usado para fazer acesso aos parâmetros usando o endereçamento baseado.

Segue-se uma nova versão da subrotina de soma que usa passagem de parâmetros de entrada pela pilha:

```
section .data
val:      dd 0
res:      dd 0
...
section .text
...
    push    dword [val]         ; res = my_add( val, 2)
    push    dword 2
    call    my_add              ; my_add( val, 2)
...                               ; eax tem o valor retornado
    mov     [res], eax
    add     esp, 8
...
my_add:
    mov     eax, [esp+4]
    mov     ebx, [esp+8]
    add     eax, ebx
    ret
```

Problemas a resolver

Todos os programas indicados a seguir devem ser testados com o ddd. Além da subrotina pedida, o seu programa deve incluir um programa principal que chame a subrotina

A.1 Programe em assembly NASM uma subrotina que retorna o quadrado do parâmetro de entrada.

```
int quadrado (int n);
```

A subrotina recebe o parâmetro de entrada em `eax` e deve retornar o resultado em `eax`. Teste com um programa principal que afecta a variável `res` com o quadrado de `x`. Estas duas variáveis estão declaradas na secção `.data` do seu programa.

A.2 Programe uma subrotina que retorna 1 se o parâmetro de entrada é par e 0 se é impar.

```
int par (int n);
```

A subrotina recebe o parâmetro de entrada em `eax` e deve retornar o resultado em `eax`. Teste com um programa principal que afecta a variável `res` com o resultado de `par(x)`. Estas duas variáveis estão declaradas na secção `.data` do seu programa.

Faça uma nova versão deste programa em que o parâmetro de entrada é passado pela pilha.

A.3 Programe uma subrotina que calcula o comprimento de uma string.

```
int my_strlen (char *s);
```

A subrotina recebe o parâmetro de entrada em `eax` e deve retornar o resultado em `eax`. Teste com um programa principal que afecta a variável `len` com o resultado de `my_strlen(s1)`. A cadeia de teste `s1` está declarada na secção `.data` do seu programa.

A.4 Recorde a Cifra de César, programada em C no exercício 6 da aula prática 3. Desenvolva duas subrotinas em assembly que realizam a codificação da Cifra de César. Admita que a cadeia de caracteres originais apenas inclui letras maiúsculas. Os parâmetros de entrada devem ser passados pela pilha.

```
void cifraCesar (char *s, char *d, int size);  
void decifraCesar(char *s, char *d, int size);
```

A.5 Programe uma subrotina em assembler para somar os elementos de um vector (soma_vector), que tem como parâmetros o endereço de um vector de inteiros e o número de elementos do vector. Utilize passagem de parâmetros pela pilha

```
int soma_vector (int *v, int size);
```

A.6 Programe a função **maiuscula**, que transforma todas as letras minúsculas da string **s** em maiúsculas.

```
void maiuscula (char *s);
```

B. Chamadas ao sistema

Seguidamente é apresentado um programa já conhecido

```
global _start

        SYS_EXIT    equ    0x01

section .text
_start:  nop
        mov     eax, SYS_EXIT    ; exit(5)
        mov     ebx, 5
        int     LINUXCALL
```

Como pode observar o programa termina sem fazer nada. De facto, as três linhas de código correspondem à invocação da chamada ao sistema EXIT que termina o programa.

Para já não é possível explicar em detalhe o que se está a passar, uma vez que a explicação da instrução *int LINUXCALL* faz parte da matéria teórica a dar mais á frente na cadeira. Uma explicação simplificada é essa instrução é uma invocação dos serviços do sistema; no caso do LINUX essa invocação é semelhante à chamada de uma subrotina em que os parâmetros são passados nos registos:

EAX: identificação da chamada ao sistema feita

0x01 exit

0x03 read

0x04 write

...

EBX: 1º parâmetro

ECX: 2º parâmetro

EDX: 3º parâmetro

No exemplo é invocada a chamada ao sistema com o código 0x01 que é *exit()*. Essa chamada tem um parâmetro (passado em *ebx*) que é o valor que o programa que termina retorna ao programa que o lançou.

Neste caso, o programa que lançou o *prog1* foi o *shell*. Na *shell* pode-se consultar o valor retornado ao shell na variável de ambiente *\$?*. Assim:

```
prompt> ./prog1
prompt> echo $?
5
prompt>
```

O exemplo seguinte envolve também o uso de chamadas ao sistema e de mais algumas pseudo-instruções do NASM.

```
global _start

SYS_EXIT    equ    0x01
SYS_WRITE   equ    0x04
LINUXCALL   equ    0x80

section .data ; marca que indica o início do zona de dados inicializados
mesg:  db    'Hello world!',0x0a
MESGSZ equ 13

section .text
_start:
    mov     eax, SYS_WRITE    ; write(1, mesg, 13)
    mov     ebx, 01
    mov     ecx, mesg
    mov     edx, MESGSZ
    int     LINUXCALL

    mov     eax, SYS_EXIT     ; exit(5)
    mov     ebx, 5
    int     LINUXCALL
```

Edite e execute este programa. Os passos a realizar são os mesmos do programa anterior.

C. Um conjunto de subrotinas para fazer chamadas ao sistema

Como se viu nos programas até agora usados, há um conjunto de funcionalidades que vão sendo usadas repetidas vezes (chamada ao sistema exit, chamada para escrever uma cadeia no terminal, etc). Os nossos programas tornar-se-ão mais simples se tivéssemos um conjunto de subrotinas para fazer chamadas ao sistema

Considere-se o código seguinte:

```
global sysWrite, sysRead, sysExit0, putchar, getchar

;chamadas ao sistema de operação (Linux)
SYS_EXIT equ 1
SYS_READ equ 3
SYS_WRITE equ 4

;INTR de chamada ao sistema
LINUXCALL equ 0x80

; sysWrite - write bytes to standard output channel
; ecx <- address where write from
; edx <- number of bytes to write
sysWrite:
    push eax
    push ebx
    mov ebx, 1 ; write to standard output
    mov eax, SYS_WRITE
    int LINUXCALL
    pop ebx
    pop eax
    ret

; sysRead - read bytes from standard input channel
; ecx <- address where to read
; edx <- max bytes to read
sysRead:
    push eax
    push ebx
    mov ebx, 0 ; read standard input
    mov eax, SYS_READ
    int LINUXCALL
    pop ebx
    pop eax
    ret

; sysExit0 - terminate program (return 0 to OS)
sysExit0:
    push eax
    push ebx
    mov ebx, 0
    mov eax, SYS_EXIT
    int LINUXCALL
    hlt ; in case something goes wrong....

;=====
```

```
; putChar - write 1 char to standard output
; al <- char to write
putChar:
```

```
    push ecx
    push edx
    push eax    ; al has ASCII to write
    mov ecx, esp
    mov edx, 1  ; write just 1 char
    call sysWrite
    pop eax
    pop edx
    pop ecx
    ret
```

```
; getChar - read 1 char from standard input
; al <- returns readed char (eax is changed)
getChar:
```

```
    push ecx
    push edx
    push eax    ; space where to read
    mov ecx, esp
    mov edx, 1  ; read just 1 char
    call sysRead
    pop eax          ; put char in al
    pop edx
    pop ecx
    ret
```

Salve este código num ficheiro chamado *io.asm*. Como se pode ver, este código implementa um conjunto de operações `sysWrite`, `sysRead`, `sysExit0`, `putChar`, `getChar` que declara como públicas (directiva *global*). Essas funções podem ser chamadas do seu programa que fica agora muito mais simples.

```
extern sysExit0, sysWrite
global _start
LF equ 0xA    ; LineFeed ascii code (new line)

section .data
    mesg:      db    'Hello world!', LF
    MESGSZ equ 13

section .text
_start:
    mov ecx, mesg
    mov edx, MESGSZ
    call sysWrite

    call sysExit0
```

Note que o programa se tornou muito simples quando comparado com o que foi antes apresentado e tem funcionalidade semelhante. Note também a pseudo-instrução *extern* que indica ao assembler que os símbolos *sysExit0* e *sysWrite* estão definidos noutra ficheiro.

Para obter o executável é preciso fazer:

```
nasm -f elf32 -g io.asm
nasm -f elf32 -g prog.asm
ld -o prog prog.o io.o
./prog
```

O **Ligador (linker ou link editor)**: é um programa de sistema que permite desenvolver programas constituídos por módulos separados. Para que esses módulos possam ser desenvolvidos separadamente é preciso que um módulo possa fazer referência a símbolos (código ou dados) definidos noutro módulo. Para esse efeito, os símbolos de um programa podem ter um dos seguintes qualificativos:

- Público: se esse símbolo puder ser referenciado de um outro módulo. No NASM este efeito é conseguido declarando o símbolo como *GLOBAL*
- Externo: este qualificativo informa o *assembler* que não deve procurar a definição do símbolo no módulo que está a processar. A definição do símbolo ficará para mais tarde. No NASM este comportamento é obtido quando o símbolo é declarado como *EXTERN*

Quem faz a resolução dos símbolos externos é o **ligador**. Este programa vai associar os símbolos externos de um módulo a símbolos públicos de outros módulos. Esta associação de nomes a valores (endereços) é feita pelo ligador permitindo a concatenação dos diferentes ficheiros **objecto** (módulos) num único ficheiro **executável**. Este será o programa final completo caso todas as resoluções dos nomes externos tenham sucesso. No caso do LINUX o ligador é um programa com o nome ld.

Problema a resolver

Faça uma versão modificada do programa da cifra de César (A-4). Dada uma cadeia de caracteres *s1* definida na área .data, o programa deve imprimir:

- a cadeia original *s1*
- a cadeia *s2* cifrada que resultou da aplicação da subrotina *cifraCesar* a *s1*
- a cadeia *s3* que resulta da aplicação da subrotina *decifraCesar* a *s2*

Naturalmente as cadeias *s1* e *s3* devem ser iguais ...