Assembly na arquitetura IA-32 com NASM no Linux

Prof. Dr. Luciano José Senger

1 Introdução

A Figura 1 mostra os registradores principais para a arquitetura `IA-32`. Essa arquitetura trabalha com palavras de 32 bits e estende os registradores de 16 bits (da arquitetura do 8088) para registradores de 32 bits. Por exemplo, o registrador `AX` é estendido para o registrador `EAX`.

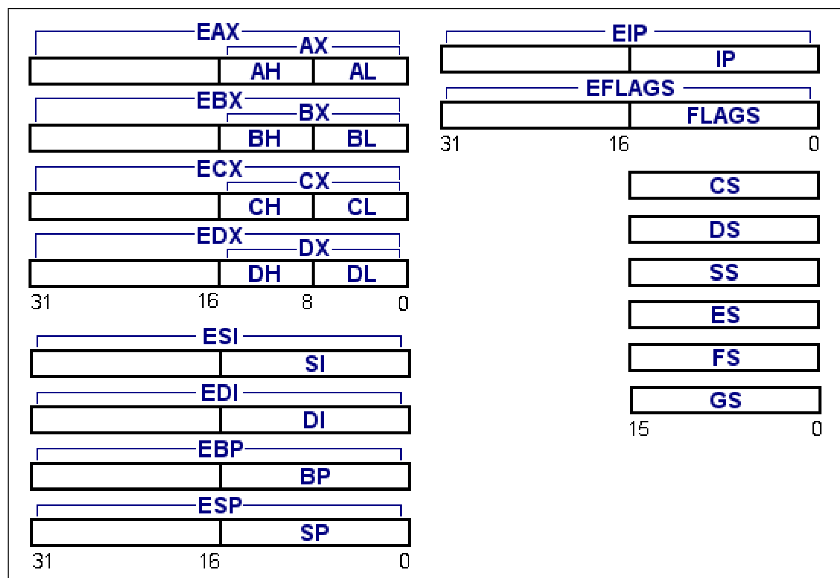

Figura 1: Registradores IA-32

2 Montagem do programas em Assembly no Linux

O quadro a seguir mostra os comando para a montagem e linkedição de programas no Linux, para um programa chamado de `program`:

Montagem e linkedição

```
nasm -f elf -o program.o program.asm

ld -o program program.o
```

Note que o comando `-f` especifica o formato do arquivo executável, que neste caso será o `elf`, padrão do sistema Linux.

Quando são usadas bibliotecas externas (p.e. stdio, stdlib), a linkedição é feita da seguinte forma:

Linkedição

```
ld -dynamic-linker /lib/ld-linux.so.2 -lc -o program program.o
```

O quadro a seguir mostra um exemplo de programa em assembly para o Linux. Note que estão sendo usados registradores de 32 bits. *Responda*:

1. Qual é o registrador utilizado pelo `kernel` do Linux para selecionar a função de escrita de *string* em tela?
2. Ao contrário do DOS, que trabalha com o serviço `21h`, qual é o número de serviço do Linux?
3. Para que serve a diretiva $, encontrada na seção de dados do programa?

Montagem

```
section .data
	hello:     db 'Hello world!',10    ; 'Hello world!' plus a linefeed character
	helloLen:  equ $-hello             ; Length of the 'Hello world!' string
	                                   ; (I'll explain soon)

section .text
	global _start

_start:
	mov eax,4            ; The system call for write (sys_write)
	mov ebx,1            ; File descriptor 1 - standard output
	mov ecx,hello        ; Put the offset of hello in ecx
	mov edx,helloLen     ; helloLen is a constant, so we don't need to say
	                     ;  mov edx,[helloLen] to get it's actual value
	int 80h              ; Call the kernel

	mov eax,1            ; The system call for exit (sys_exit)
	mov ebx,0            ; Exit with return code of 0 (no error)
	int 80h
```

Esse arquivo, se salvo com o nome `helloWorld.asm`, pode ser montado e linkeditado da seguinte forma:

Montagem e linkedição

```
nasm -f elf -o helloWorld.o helloWorld.asm
ld -o helloWord helloWorld.o
```

Empregando o comando `nm`, pode-se obter a lista de símbolos do programa:

o comando nm

```
nm helloWorld.o
```

A saída deste comando é:

saída do comando nm

```
080490b1 A __bss_start
080490b1 A _edata
080490b4 A _end
08048080 T _start
080490a4 d hello
0000000d a helloLen
```

Responda : Oque significam as informações obtidas? (dica: use o comando `man nm`) para consultar a página de manual do comando `nm`.

A seguir, tem se o resultado da execução do comando:

comando objdump

```
objdump -d helloWorld
```

Pode-se verificar o código em assembly (nesse caso o `GAS`, padrão em sistemas UNIX) e os códigos em linguagem de máquina para as instruções:

saída do comando objdump

```
helloWorld:     file format elf32-i386

Disassembly of section .text:

08048080 <_start>:
8048080:       b8 04 00 00 00          mov    $0x4,%eax
8048085:       bb 01 00 00 00          mov    $0x1,%ebx
804808a:       b9 a4 90 04 08          mov    $0x80490a4,%ecx
804808f:       ba 0d 00 00 00          mov    $0xd,%edx
8048094:       cd 80                   int    $0x80
8048096:       b8 01 00 00 00          mov    $0x1,%eax
804809b:       bb 00 00 00 00          mov    $0x0,%ebx
80480a0:       cd 80                   int    $0x80
```

Responda: Quais são as principais diferenças da sintaxe do `GAS` em relação ao `NASM`?

2.1 Obtendo os argumentos da linha de comando

A pilha tem uma função importante em sistemas Linux: a comunicação do sistema operacional com o programa em execução. Através da pilha, os argumentos são passados ao programa para o processamento. Considere a execução do programa abaixo `imprimeMsg` que recebe como parâmetro a string `Ola`:

```
imprimeMsg Ola 5
```

Nesse caso a, configuração da pilha é a seguinte:

```
4          ; quantidade de argumentos
imprimeMsg; nome do programa
Ola        ; primeiro argumento
5          ; segundo argumento
```

O programa `imprimeMsg` está listado a seguir:

O programa imprimeMsg

```
section .data
        lineBreak  db 10
section .text
        global _start


_start:
        mov ebp, esp
        mov eax,4
        mov ebx,1
        mov ecx, [ebp+8]
        mov edx, 3
        int 80h              ; Call the kernel

        mov eax,4
        mov ebx,1
        mov ecx, lineBreak
        mov edx, 1
        int 80h

        mov     eax,1
        mov     ebx,0
        int     80h             ; Exit
```

Exercício: Modifique o programa anterior para que o mesmo imprima em tela todos os parâmetros recebidos pelo programa, e não apenas o primeiro argumento. Salve este programa como `imprimeMsg2.asm`.

2.2 Verificando em tempo de execução as chamadas de sistema

O comando `strace` inicia a execução do programa passado como parâmetro e mostra em tela uma listagem de todas as chamadas de sistema invocadas pelo programa: Assim, o comando:

O comando strace

```
strace ./imprimeMsg Olateste 5
```

produz como resposta:

Listagem obtida com o comando strace

```
execve("./imprimeMsg", ["imprimeMsg", "Olateste", "5"], [/* 33 vars */]) = 0
write(1, "Olate", 5Olate)                    = 5
write(1, "\n", 1
)                        = 1
_exit(0)                                = ?
```

2.3 Utilização de macros

Note no exemplo anterior a repetição no uso da `syscall write`. Através do uso de macro, fragmentos de código frequentemente usados podem ser definidos como `macros`, inclusive com parâmetros. *Responda*: Quais são as vantagens e desvantagens de `macros` em relação aos procedimentos?

resposta

Um exemplo do uso de macros para facilitar o acesso à `syscalls` é mostrado a seguir . As macros são definidas nas linhas 16 e 27. Um exemplo do uso no programa principal pode ser visto na linha 42.

macros

```
section .data
prompt_str  db   'Enter your name: '
STR_SIZE  equ  $ - prompt_str

greet_str  db  'Hello '


GSTR_SIZE  equ  $ - greet_str


section .bss

; Reserve 32 bytes of memory
buff  resb  32

; A macro with two parameters
; Implements the write system call
%macro write 2
mov   eax, 4
mov   ebx, 1
mov   ecx, %1
mov   edx, %2
int   80h
%endmacro


; Implements the read system call
%macro read 2
mov   eax, 3
mov   ebx, 0
mov   ecx, %1
mov   edx, %2
int   80h
%endmacro


section .text

global _start

_start:
write prompt_str, STR_SIZE
read  buff, 32

; Read returns the length in eax
push  eax

; Print the hello text
write greet_str, GSTR_SIZE

pop   edx

; edx  = length returned by read
```

```
write buff, edx

_exit:
mov   eax, 1
mov   ebx, 0
int   80h
```

3 Procedimentos

Na linguagem Assembly, procedimentos são invocados através da instrução `CALL`, seguida de um rótulo. Por exemplo, `CALL R1` desvia a execução para o rótulo `R1`. O endereço de retorno no término do procedimento é armazenado na pilha, e através da instrução `RET`, o retorno é executado.

Existem três formas de passar parâmetros para procedimentos: registradores, variáveis globais e através da pilha. Registradores e variáveis globais não são adequados, se for necessário a passagem de um número variável de parâmetros ou quando é necessário que o procedimento seja invocado por ele mesmo, em chamadas recursivas. Assim, geralmente a passagem de parâmetros pela pilha é mais utilizada, tanto em programas escritos em Assembly como linguagens de alto nível, como C, Fortran e Pascal.

Como exemplo, suponha que um procedimento chamado `Imul2` calcula a multiplicação entre dois inteiros que foram inseridos na pilha e retorna o resultado em `EAX`:

```
push VarA ; dword
push VarB ; dword
call Imul2
add ESP, 8
```

Assume-se que o procedimento não afeta a pilha, de forma que ao fim de sua execução, o ponteiro da pilha `ESP` referencia a mesma posição antes da chamada. Isso explica a operação `add ESP,8`, que é necessária para restaurar o valor inicial do `ESP`, pois cada uma das variáveis têm 16 bits de tamanho. Esse método é chamado como *restauração da pilha pelo código que invocou a chamada* (*stack restored by the caller*) e utilizado pela maioria dos compiladores C/C++. A convenção adotada na linguagem C/C++ é empilhar os parâmetros na ordem inversa em que aparacem na declaração da função (da direita para a esquerda).

O código do procedimento pode ser escrito então:

```
Imul2:
        push ebp
        mov  ebp, esp
        mov  eax, [ebp+12]
        Imul eax, [ebp+8]
        pop ebp
        ret
```

O primeiro parâmetro está na posição `ESP+8`. A longo da execução da função/procedimento o valor de `ESP` pode variar. Por isso, utiliza-se um outro registrador, `EBP`, para referenciar a base da pilha durante toda a execução da função/procedimento. A porção da pilha referente a cada função/procedimento é chamada de *registro de ativação*. A Figura 2 mostra o conteúdo da pilha para esse exemplo.

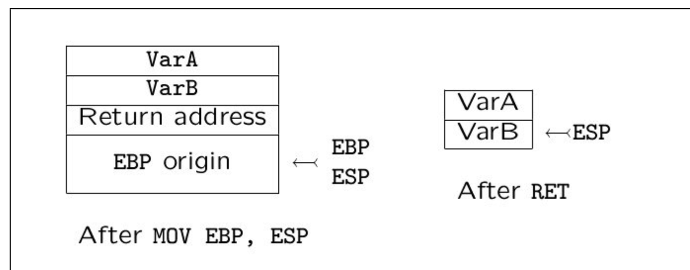

Figura 2: Exemplo de registros de ativação

Assim, `ebp` referencia sempre a base do registro de ativação corrente. Na medida que os procedimentos são invocados, a pilha de execução cresce. Além dos endereços de retorno, é comum cada procedimento usar a pilha para armazenar variáveis locais. A figura 3 ilustra os registros de ativação para uma chamada encadeada de procedimentos.

Seguindo tais convenções, pode-se unir o código escrito em Assembly com código escrito na linguagem C. Tal ligação é comum no desenvolvimento de software básico, como por exemplo o software de um sistema operacional.

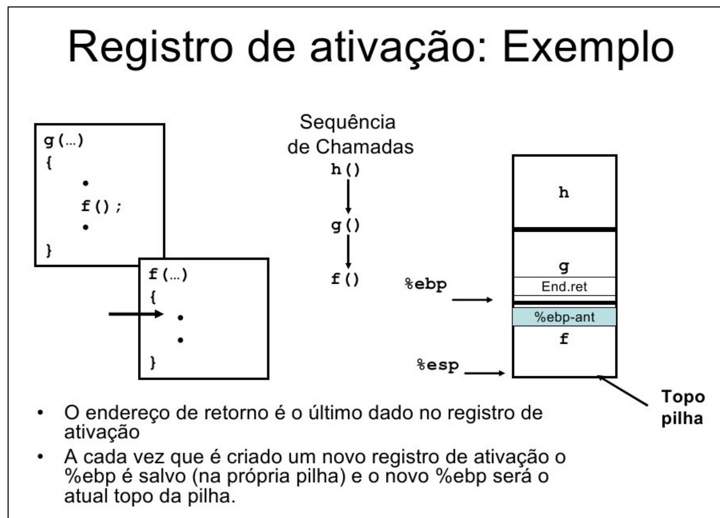

Figura 3: Exemplo de registros de ativação

4 Utilização de bibliotecas externas

O quadro a seguir mostra um exemplo de utilização de rotinas externas:

Exemplo de utilização de bibliotecas externas

```
section .data

; Command table to store at most
;  10 command line arguments
cmd_tbl:
%rep 10
dd 0
%endrep

section .text

global _start

_start:
; Set up the stack frame
mov   ebp, esp
; Top of stack contains the
;  number of command line arguments.
; The default value is 1
mov   ecx, [ebp]

; Exit if arguments are more than 10
cmp   ecx, 10
jg    _exit

mov   esi, 1
mov   edi, 0

; Store the command line arguments
;  in the command table
store_loop:
        mov   eax, [ebp + esi * 4]
        mov   [cmd_tbl + edi * 4], eax
        inc   esi
        inc   edi
        loop  store_loop

mov   ecx, edi
mov   esi, 0
```

```
extern puts

print_loop:
        ; Make some local space
        sub   esp, 4
        ; puts function corrupts ecx
        mov   [ebp - 4], ecx
        mov   eax, [cmd_tbl + esi * 4]
        push  eax
        call  puts
        add   esp, 4
        mov   ecx, [ebp - 4]
        inc   esi
        loop  print_loop

jmp   _exit

_exit:
mov   eax, 1
mov   ebx, 0
int   80h
```

Salve o programa como `readArgs.asm`. Realize a montagem e linkedição, lembrando de especificar na linkedição a biblioteca externa. *Verifique você mesmo:*

1. use os comandos `nm` e `objdump` para verificar as informações detalhas sobre o arquivo objeto;
2. use o comando `strace` para verificar as chamadas de sistema empregadas.

4.1 Diferenças entre o IA-32 e a arquitetura MIPS em relação aos procedimentos

Como a arquitetura IA-32 tem menos registradores de propósito geral, tal arquitetura não pode convencionar alguns registradores específicos para isso conforme existe na arquitetura MIPS. A quantidade menor de registradores da arquitetura IA-32 implica em uma maior utilização da pilha na passagem de parâmetros e no para salvamento de valores de registradores.