Universidade Federal do Rio Grande do Sul
Escola de Engenharia
Departamento de Sistemas Elétricos de Automação e Energia
ENG10032 Microcontroladores

Roteiro de Laboratório 04
Convenções de Chamada de Função

Prof. Walter Fetter Lages

10 de abril de 2019

1 Objetivo

O objetivo deste laboratório é explorar as convenções de chamada de subrotinas da linguagem C. Isto será feito através do desenvolvimento de rotinas em Assembly que possam ser chamadas a partir de programas em linguagem C.

2 Introdução

Em linguagens de alto-nível, os parâmetros para as funções, sub-rotinas, etc. são passados através da pilha do sistema. Cada linguagem possui certas convenções que devem ser obedecidas para esta passagem de parâmetros. Sendo obedecidas estas convenções, torna-se irrelevante em que linguagem a subrotina foi escrita. As convenções utilizadas pelas diversas linguagens diferem entre si, basicamente pelos seguintes aspectos:

- Ordem com que os parâmetros são colocados na pilha
- Se os parâmetros são passados por valor (o parâmetro é colocado na pilha) ou por referência (o endereço do parâmetro é colocado na pilha)
- Se quem limpa a pilha (remove os parâmetros) é a função que chama ou a função que é chamada

Em particular na linguagem C, a convenção é que os parâmetros são colocados na pilha por valor, da direita para a esquerda (ou seja, o último parâmetro é o primeiro a ser colocado na pilha) e quem limpa a pilha é a função que está fazendo a chamada. Estas convenções são particularmente interessantes porque

permitem que sejam implementadas facilmente funções com número variável de argumentos, com por exemplo a função `printf()`, cujo protótipo é:

```
int printf(const char *format,...)
```

Note que os ... (elipses, na terminologia de C) indicam um número variável de parâmetros. O número de argumentos com o qual a função é chamada é especificado através de um dos argumentos anteriores.

Usualmente em sistemas IA32, tem-se que o valor de retorno da função deve estar no registrador `eax`, se for de 32 bits ou `edx:eax`, se for de 64 bits, ou `st(0)`, se for ponto flutuante.

No Linux tem-se ainda que os registradores `ebp`, `esi`, `edi` e `ebx`, ou seja, os registradores de base e de índice, devem ser preservados através da chamada da subrotina (isto é, devem ser salvos e restaurados, se forem utilizados pela subrotina), enquanto os demais podem ser utilizados livremente.

Em sistemas `x86_64`, quando executando código de 64 bits, a convenção de chamada de funções é otimizada para passar os parâmetros por registrador quando a função possui poucos parâmetros. Os registradores `rdi`, `rsi`, `rdx`, `rcx`, `r8` e `r9` são utilizados para passar argumentos inteiros e ponteiros e os registradores `xmm0`, `xmm1`, `xmm2`, `xmm3`, `xmm4`, `xmm5`, `xmm6` e `xmm7` são utilizados para argumentos em ponto flutuante. Os demais argumentos são passados pela pilha. O valor de retorno da função deve estar em `rax` e os registradores `rbp` e `rbx` devem ser preservados.

3 Programa em C Chamando Rotinas em Assembly

Para demonstrar como funcionam as convenções de chamada, nesta seção será desenvolvido um programa em C para calcular a soma e a diferença de dois números inteiros. A soma e a diferença serão calculadas por rotinas implementadas em Assembly. O programa em C está na Listagem 1.

Listagem 1: Programa em C chamando rotinas em Assembly.

```
#include <stdio.h>
#include <stdlib.h>

#include <asmops.h>

int main(int argc,char *argv[])
{
        int a;
        int b;
        int s;
        int d;
```

```
        if(argc != 3)
        {
                printf("Usage:\n\t%s <int> <int>\n",argv[0]);
                return -1;
        }

        a=atoi(argv[1]);
        b=atoi(argv[2]);

        s=sum(a,b);
        d=diff(&a,&b);

        printf("%d+%d=%d\n%d-%d=%d\n",a,b,s,a,b,d);

        return 0;
}
```

Note que a soma é calculada pela função `sum()` e que a diferença é calculada pela função `diff()`. Os protótipos destas funções estão definidos no arquivo `asmops.h`, mostrado na Listagem 2:

Listagem 2: Arquivo de cabeçalho `asmops.h`.

```
#ifndef ASMOPS_H
#define ASMOPS_H

extern int sum(int a,int b);
extern int diff(int *a,int *b);

#endif
```

Na função `sum()` os parâmetros são as variáveis contendo os valores a serem somados. Já na função `diff()` os parâmetros são os endereços das variáveis (ponteiros) contendo os valores a serem somados. Usualmente se passa ponteiros para as variáveis quando se deseja que a função chamada possa alterar o conteúdo das variáveis passadas ou quando as variáveis são grandes em termos de memória, de forma que passar apenas um ponteiro é mais rápido.

A passagem de ponteiros para as variáveis é muito semelhante à passagem de parâmetros por referência, e muitas vezes é confundida com ela, mas não é exatamente igual. Na passagem por referência, a criação do ponteiro é realizada automaticamente pelo compilador, sem que seja necessária qualquer sintaxe especial para isto. Efetivamente, em C este não é o caso, já que é necessário utilizar o operador `&` para criar o ponteiro.

A implementação das funções em Assembly é mostrada na Listagem 3:

Listagem 3: Funções implementadas em Assembly.

```
        .intel_syntax noprefix

        .text

        .global sum
        .global diff

# int sum(int a,int b)
sum:
        push    ebp
        mov     ebp,esp
        push    ebx
        mov     eax,[ebp+8]     # a
        add     eax,[ebp+12]    # b
        pop     ebx
        pop     ebp
        ret

# int diff(int &a,int &b)
diff:
        push    ebp
        mov     ebp,esp
        push    ebx

        mov     ebx,[ebp+8]     # &a
        mov     eax,[ebx]       # a
        mov     ebx,[ebp+12]    # &b
        sub     eax,[ebx]       # b

        pop     ebx
        pop     ebp
        ret
```

Note que é utilizado o registrador `ebp` para acessar os dados na pilha (quando é usado este registrador ou o `esp` para indexação, por *default* é usado o segmento de pilha) e que na função `diff`, primeiro são obtidos da pilha o endereço dos parâmetros e depois os parâmetros são obtidos utilizando-se endereçamento indireto através do registrador `ebx` (quando é usado este registrador ou `si` ou `di` para indexação, por *default* é usado o segmento de dados).

Devido as convenções de chamada e à forma como foi montado o código de prefácio das funções, o primeiro parâmetro sempre estará em no endereço apontado por `ebp+8`.

4 Experimentos

1. Digite o programa em C em um arquivo denominado `callasm.c`.

2. Digite o arquivo cabeçalho em um arquivo denominado `asmops.h`.

3. Digite as rotinas em Assembly em um arquivo denominado, por exemplo `asmops.s`.

4. Digite o `Makefile` mostrado na Listagem 4. Note que as opções `--32`, para o assembler e `-m32` para o compilador, forçam a geração de código de 32 bits. Isto é necessário porque o código em Assembly está utilizando as convenções de chamada para 32 bits. Aqui isto não é estritamente necessário porque as ferramentas de desenvolvimento para a Galileo geram apenas código para 32 bits. Note que a compilação do programa em C é feita **sem** a *flag* `-O2`.

   Listagem 4: `Makefile` para gerar o executável do programa em C chamando rotinas em Assembly.

```
TARGET=callasm
CSRCS=$(TARGET).c
ASMSRCS=asmops.s

FLAGS=
INCLUDE=-I.
LIBDIR=
LIBS=

AS=$(CROSS_COMPILE)as
ASFLAGS=-gstabs -a=`echo $@ | cut -f 1 -d.`.lst --32 \
        -MD=`echo $@ | cut -f 1 -d.`.d
CC=$(CROSS_COMPILE)gcc
CFLAGS=-Wall -MMD $(INCLUDE) -g -m32
LDFLAGS=-m32

all: $(TARGET)

$(TARGET): $(CSRCS:.c=.o) $(ASMSRCS:.s=.o)
        $(CC) -o $@ $^ $(LDFLAGS)

%.o: %.s
        $(AS) $(ASFLAGS) -o $@ $<

-include $(ASMSRCS:.s=.d)

%.o: %.c
        $(CC) $(CFLAGS) -c -o $@ $<

-include $(CSRCS:.c=.d)

clean:
        rm -f *~ *.bak *.o *.d *.lst

distclean: clean
```

```
	rm -f $(TARGET)
```

5. Gere o programa executável digitando

```
make
```

6. Carrege o programa na Galileo para depuração remota com o comando:

```
gdbserver <host>:<port> callasm 1 2
```

   onde `<host>` é o nome do *host* utilizado e `<port>` é a porta TCP utilizada para comunicação da Galileo com o *host*.

7. Carregue o `kdbg` no *host* para depuração remota com o comando:

```
kdbg -r <target>:<port> callasm
```

   onde `<target>` é o nome da Galileo utilizada.

8. Carrege o código fonte do programa no `kgdb`.

9. Insira um `breakpoint` no início do programa e clique nos sinais de + a esquerda para visualizar o código em Assembly do programa.

10. Execute o programa passo-a-passo para verificar o seu funcionamento e como é feita a passagem dos parâmetros nos dois casos. Observe, particularmente o funcionamento da pilha do sistema.

11. Insira a *flag* `-O2` nas *flags* de compilação e repita os passos 5 a 10. Obs: antes de executar o `make` execute um `touch callasm.c`, para forçar a compilação já que o próprio `Makefile` não é uma dependência para gerar o executável.

12. Uma *string* em C é uma sequência de bytes com o código ASCII dos caracteres terminada por um byte como valor 0x00. Faça um programa em C para imprimir na tela em letras maiúsculas uma *string* passada na linha de comando. A conversão para letras maiúsculas e a impressão na tela deve ser feita por uma função implementada em Assembly. Dica: O ASCII de uma letra maiúscula diferencia-se da respectiva minúscula por apenas 1 bit.

A Chamadas do *Kernel* do Linux (`int0x80`)

A.1 Write

Escreve caracteres de um arquivo

parâmetros:

eax: código da função=4

ebx: descritor do arquivo[1]

ecx: ponteiro para o *buffer*

edx: tamanho do buffer

retorno:

eax: número de bytes escritos

erros:

eax: EAGAIN, EBADF, EFAULT, EINTR, EINVAL, EIO, ENOSPC, EPIPE

[1] stdout é o descritor 1, stderr é o descritor 3 e estão sempre abertos.