Arquitetura de Computadores 2017-18

Ficha 7

Tópicos: *Subrotinas e convenções de chamada em assembly. Funções recursivas. Ligação de código assembly e C.*

1. Código produzido pelo compilador

Considere o seguinte exemplo de programa em C. Verifique o que este faz, compile-o e execute-o. *Nota: Se estiver num sistema Linux de 64bits compile em 32bits acrescentando a opção "-m32" na compilação.*
Exemplo: ***`cc -m32 -o prog prog.c`***

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

int calc( int a ) {
      return a + 1;
}

int main( int argc, char *argv[] ) {
      int x, y;
      if ( argc!=2 ) {
            fprintf(stderr,"use: %s <int num>\n", argv[0] );
            return 1;
      }
      x = atoi(argv[1]);
      y = calc( x );
      printf("y=%d\n", y );
      return 0;
}
```

Pode obter o código produzido pelo compilador usando a opção "-S" na compilação (`cc -m32 -S prog.c`), que irá produzir um ficheiro "prog.s". Também pode obter o código presente em qualquer executável usando o comando `objdump`. Como visto antes, se o seu programa executável se chamar "prog" use: `objdump -d prog`

Procure na listagem produzida por qualquer um dos comandos anteriores o código das funções *main* e *calc*. Se tiver compilado com informação de *debug* (-g) pode obter ainda a relação entre cada linha do código fonte em C e o respetivo *assembly*, com o comando: `objdump -dS prog`

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 de biblioteca.

2. Ligação de C com *assembly*

A partir do exemplo anterior vamos implementar a função calc diretamente em *assembly* e ligar esta ao programa C. Crie o ficheiro "calc.s" com a seguinte subrotina (correspondente à implementação da função calc anterior):

```
/* calc.s
 *    for Linux (32bits)
 */
.text
.global calc

calc:
  push %ebp
  mov %esp, %ebp
  mov 8(%ebp), %eax      # get argument (EBP+8)
  add $1,%eax
  pop %ebp
  ret
```

(na realidade este código poderia ser muito mais simples; consegue simplificar?)

No seu programa C deixa de ter a função *calc* que tinha antes. Apague-a! Deve apenas indicar que a função *calc* está implementada noutro ficheiro, declarando apenas o seu protótipo do seguinte modo:

```
extern int calc( int a );
```

Pode usar o comando 'as' para "assemblar" e, depois, ligar o código máquina (também chamado binário, ou 'objecto') ao programa em C:

```
as -g -o calc.o calc.s
cc -g -o prog prog.c calc.o
```

(se estiver em Linux/64bits use como primeiro comando: `as --32 -g -o calc.o calc.s`)

Em alternativa pode usar o compilador de C para efetuar todos esses passos usando:

```
cc -g -o prog prog.c calc.s
```

Confirme que o programa continua a funcionar corretamente, inclusive no *debugger* (gdbtui). Veja também o código do executável usando o comando *objdump* como antes.

3. Fibonacci Recursivo

Copie o seguinte programa C para o editor de texto no seu computador, dando-lhe o nome "fibonacci.c", compile-o e execute-o. O programa recebe uma *string* na linha de comandos (contendo um conjunto de dígitos numéricos) que converte num inteiro. Depois chama a função *fibonacci()* e imprime o resultado no ecrã.

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

int fibonacci( int n ) {
    if (n == 0) return 0;
    if (n == 1) return 1;
    return fibonacci (n - 1) + fibonacci (n - 2);
}

int main( int argc, char *argv[] ) {
    if (argc != 2) {
        fprintf( stderr, "use: %s <int_num>\n", argv[0] );
        return 1;
    }
    int n = atoi( argv[1] );
    int f = fibonacci ( n );
    // print the results
    printf( "Fibonacci(%d)=%d\n", n, f );
    return 0;
}
```

Implemente agora a função *fibonacci* em *assembly* num ficheiro "`fib.s`". Crie o executável e experimente-o.

4. Ligação *assembly* com C

Dadas duas *strings*, cada uma representando um valor inteiro, pretende-se implementar em *assembly* uma função que imprime o resultado da soma desses inteiros, de acordo com o seguinte protótipo C:

```
void  printSoma( char *str1, char *str2 );
```

Use a função `atoi` da biblioteca de C para converter cada *string* para o respetivo inteiro e recorra a `printf` para escrever o resultado. Use como ponto de partida o código seguinte:

```
.text
.global printSoma

## @arg1: string com digitos (representado um numero)
## @arg2: string com digitos (representado um numero)

printSoma:
      push %ebp
      mov %esp, %ebp

## complete o codigo ##
## objetivo 1: converter o primeiro argumento para um inteiro

      call atoi

## complete o codigo ##
## objetivo 2: converter o segundo argumento para um inteiro

      call atoi

## complete o codigo ##
## objetivo 3: realizar a soma dos dois inteiros
## objetivo 4: imprimir o resultado da operação

      call printf

## complete o codigo ##
## objetivo 5: garantir que a pilha está no mesmo estado
##             em que estava quando printSoma foi invocada

      pop %ebp
      ret
```

Teste escrevendo em C um programa que obtém duas *strings* com valores numéricos, à semelhança do `main` de programas anteriores e que chama a sua função para escrever a respetiva soma. Exemplo:

```
#include <stdio.h>

extern void  printSoma( char *str1, char *str2 );

int main( int argc, char*argv[] ) {
   if (argc != 3) {
      fprintf( stderr, "Usage: %s <int_num> <int_num>\n", argv[0] );
      return 1;
   }
   printSoma( argv[1], argv[2] );
   return 0;
}
```

Para compilar e ligar use o compilador de C. Exemplo:

```
      cc -g -o prog prog.c printSoma.s
```

4. Mais informação

Convenção de chamada em Linux/ABI 32bits: para efetuar a chamada de uma função, em C, usa-se a convenção normalmente chamada de **`cdecl`**, que é a seguinte:

1. os argumentos para a função são colocados na pilha, da "direita para a esquerda"
2. é efectuado o call;
3. na função (subrotina) não se podem alterar os registos gerais excepto %eax, %ecx e %edx (se quiser usar outros, tem de os salvar na pilha e repor o valor original antes do retorno);
4. o resultado fica em %eax.

Mais informação nas secções 2.1 e 2.2 de:
System V Application Binary Interface, Intel386, version 1, edited by: H.J. Lu, David L Kreitzer, Milind Girkar, Zia Ansari. Intel Open Source: http://01.org/sites/default/files/file_attach/intel386-psabi-1.0.pdf
Ver ainda a página seguinte.

departamento de informática
FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

Intel x86 (IA32) Assembly Language Cheat Sheet

Suffixes: b=byte (8 bits); w=word (16 bits); l=long (32 bits). Optional if instruction is unambiguous.

Operands: immediate/constant (not as *dest*): $10, $0xff ou $0b01101 (decimal, hex or bin)
32-bit registers: %eax, %ebx, %ecx, %edx, %esi, %edi, %esp, %ebp
16-bit registers: %ax, %bx, %cx, %dx, %si, %di, %sp, %bp
8-bit registers: %al, %ah, %bl, %bh, %cl, %ch, %dl, %dh
direct addr: (2000) or (0x1000+53)
indirect addr: (%eax) or 16(%esp) or 200(%edx, %ecx, 4)

No te that it is not possible for **both** *src* and *dest* to be memory addresses.

Instruction	Effect	Examples
Copying Data		
mov *src,dest*	Copy src to dest	mov $10,%eax movw %ax,(2000)
Arithmetic		
add *src,dest*	dest = dest + src	add $10, %esi
sub *src,dest*	dest = dest – src	sub %eax,%ebx
cmp *src,dest*	Compare using sub (dest is not changed)	cmp $0,%eax
inc *dest*	Increment destination	inc %eax
dec *dest*	Decrement destination	decl (0x1000)
Bitwise and Logic Operations		
and *src,dest*	dest = src & dest	and %ebx, %eax
test *src,dest*	Test bits using and (dest is not changed)	test $0xffff,%eax
or *src,dest*	dest = src | dest	or (0x2000),%eax
xor *src,dest*	dest = src ^ dest	xor $0xffffffff,%ebx
shl *count,dest*	dest = dest << count	shl $2,%eax
shr *count,dest*	dest = dest >> count	shr $4,(%eax)
sar *count,dest*	dest = dest >> count (preserving signal)	sar $4,(%eax)
Jumps		
je/jz *label*	Jump to label if dest == src /result is zero	je endloop
jne/jnz *label*	Jump to label if dest != src /result not zero	jne loopstart
jg *label*	Jump to label if dest > src	jg exit
jge *label*	Jump to label if dest >= src	jge format_disk
jl *label*	Jump to label if dest < src	jl error
jle *label*	Jump to label if dest <= src	jle finish
ja *label*	Jump to label if dest > src (unsigned)	ja exit
jae *label*	Jump to label if dest >= src (unsigned)	jae format_disk
jb *label*	Jump to label if dest < src (unsigned)	jb error
jbe *label*	Jump to label if dest <= src (unsigned)	jbe finish
jz/je *label*	Jump to label if all bits zero	jz looparound
jnz/jne *label*	Jump to label if result not zero	jnz error
jmp *label*	Unconditional jump	jmp exit
Function Calls / Stack		
call *label*	Call (Push eip and Jump)	call format_disk
ret	Return to caller (Pop eip and Jump)	ret
push *src*	Push item to stack	pushl $32
pop *dest*	Pop item from stack	pop %eax

Directives (examples):

.data – data section (global variables)
.int – 32bits space(s) for integer value(s)
.ascii – char sequence
.global *label* -- export *label* symbol/address
.text – text section (code)
.comm *label*, *length* – length bytes space

Functions Linux/32bits:

caller:
- push args (right to left)
- call function
- free stack space used with args

C types:
char 1 byte
short 2 bytes
int, float, long and *pointer* 4 bytes
double 8 bytes

callee (function):
- initialise: push %ebp
  mov %esp, %ebp
  sub $4, %esp #space for local var.
- use ebp based address, e.g.: movl 8(%ebp), %eax
- result at %eax
- finalise: mov %ebp, %esp #free local var.
  pop %ebp
  ret