NOVA SCHOOL OF SCIENCE & TECHNOLOGY

Arquitetura de Computadores 2021/22

Trabalho Individual 2 (Individual Assignment 2)

Due: April 28, 2022, 23:59

This homework consists of two individual programming exercises in assembly Linux 32 bits (IA32, also named i386 or x86). You can discuss general doubts with colleagues but the solution and the code writing should be strictly individual. All solutions will be automatically compared and plagiarism cases will be punished in accordance with the regulations.
Your solutions are to be submitted to DI's Mooshak (http://mooshak.di.fct.unl.pt/~mooshak/). You are limited to 10 submissions for each exercise (more will be ignored!). The OS is a Linux and your program is compiled with the following command to ensure 32bits architecture is used (in case of errors or warnings your program will fail!):

```
as --32 -o prog.o prog.s
ld -m elf_i386 -o prog prog.o
```

1.Counting words (50%)

Consider a program in assembly where a string is defined and its goal is to count how many words there are in that string. The word separator in the original sentence is only the blank space, and for simplification, it is assumed that the original sentence only contains letters. However, there may be more than one blank space separating the words. The result of the count should be left in the long word *total* and returned to the shell.
For instance, consider the following representations for the data section in your program:

Example:

```
.data       # data section (variables)
  msg: .ascii " Spring  begins in   March "
  MSGLEN = (. - msg)
  total: .int 0
```

The total number of words in this example is 4 and this is the value that it is stored in the variable *total* as well as returned to the shell with the code

```
movl    total,%ebx
movl    $EXIT,%eax        # pedir o exit ao sistema
int     $LINUX_SYSCALL    # chama o sistema
```

After the execution ends and on returning to the shell prompt, the value can be obtained by executing the command `echo $?` (Note: the `echo` command must be executed immediately after the program ends; in case you execute another command in between the result does not represent the return of your program). For instance, for the example above the result will be

```
$ echo $?
4
```

2.Summing elements with a stride (50%)

Consider a program with the definition of a vector of integers v. The goal is to sum the vector's elements with a stride, i.e. a gap is considered on summing the elements. For instance, if the stride is

1 the program sums all numbers in the vector, i.e. it sums elements v[0] + v[1] + v[2] + v[3], etc., of the vector. If the stride is two, the program will sum elements with even indices i.e. v[0] + v[2] + v[4] etc. If the stride is 3, the program sums elements v[0]+v[3]+v[6]+v[9], etc. The sum result has to be put in the long word *total* and returned as a result to the shell.
For instance, consider the following representations for the data section in your program:

Example:

```
.data       # data section (variables)
  v: .int -1, 5, 1, 1, 4
  len = (. - msg)/4
  stride: .int 2
  total: .int 0
```

In this case, since the stride is 2, the program will sum elements v[0] + v[2] + v[4] = -1+1+4 = 4, and the variable total will be equal to 4. In case the stride is 3 the sum is v[0] + v[3] = -1+1=0, and in case the stride is 5 the sum will be only v[0]= -1.

As in the first problem, the value that it is stored in the variable *total* is also returned to the shell with the following code,:

```
movl    total,%ebx
movl    $EXIT,%eax        # pedir o exit ao sistema
int     $LINUX_SYSCALL    # chama o sistema
```

After the execution ends and on returning to the shell prompt, the value can be obtained by executing the command `echo $?`

NOVA SCHOOL OF SCIENCE & TECHNOLOGY

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)

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

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