

## Teste 1A de Arquitetura de Computadores 08/05/2021

### Duração 2h

- Teste sem consulta e sem esclarecimento de dúvidas
- A detecção de fraude conduz à reprovação de todos os envolvidos

Nº:

Nome:

**Q1 – 1,5 valores** Considere um sistema computacional em que cada posição de memória tem 12 bits. Responda às perguntas seguintes sobre o conteúdo que pode ser guardado nessa posição de memória **M**. As suas respostas não precisam de indicar um valor, podendo ser uma expressão numérica que inclui uma potência de 2.

- a) Se a sequência de bits guardada na posição de memória **M** for interpretada como um número inteiro sem sinal, qual o menor e o maior número que pode ser guardado em **M**?
- b) Suponha que para representar números inteiros com sinal é usada a representação em complemento para 2. Indique o valor dos 12 bits de **M** quando nela é armazenado -3 (base 10)? Apresente os passos que deu até chegar à resposta.
- c) Suponha que para representar números inteiros com sinal é usada a representação em complemento para 2. Qual é o maior e o menor inteiro com sinal que pode ser armazenado em **M**?

**Q2- 2.0 valores** Considere um CPU que tem uma unidade aritmética e lógica (ALU) em que as duas entradas têm 32 bits e a saída tem 32 bits; os valores dos operandos e resultado estão em complemento para 2. A UAL tem associadas duas *flags* CF (*Carry Flag*) e OF (*Overflow Flag*).

- a) A ALU acabou de fazer uma soma de dois valores **U1** e **U2** que são o conteúdo de duas variáveis em **C** declaradas como *unsigned*. A CF está a 1. O que significa significa em termos do resultado produzido pela ALU?
- b) A ALU acabou de fazer uma soma de dois valores **V1** e **V2** que são o conteúdo de duas variáveis em **C** declaradas como *signed*. A OF está a 1. O que é que isto significa em termos do resultado produzido pela ALU?

**Q3- 2.0 valores** Na representação de números reais em precisão simples IEEE Floating Point Standard um número real é representado em 32 bits (o bit 31 é o mais significativo e o bit 0 é o menos significativo) com a seguinte interpretação:

- Bit 31: sinal – 0 maior ou igual a zero, 1 menor do que zero
- Bits 30 a 23 (8 bits): expoente **E**. Sendo **E** um inteiro sem sinal codificado nestes bits, o valor real do expoente é  $E - 127$
- Bits 22 a 0 (23 bits): mantissa. Sendo a configuração dos bits da mantissa  $xxxxx...xxx_2$ , o valor efetivo da mantissa é  $1.xxxx...xx_2$

Considere o número real representado por  $1\ 10000001\ 100000000000000000000000$ . Preencha a seguinte tabela. Note que a 4ª coluna da tabela só será considerada se a 1ª, 2ª e 3ª colunas estiverem preenchidas

Sinal(+ ou -)	Expoente (base 10)	Mantissa (base 2)	Valor representado (base 10)

**Q4- 1.5 valores** Considere a seguinte sequência de instruções

```
movl $5, %eax
cmpl $6, %eax    5<=6
jle 11
movl $5, %ebx
jmp 12
```

```
11: movl $6, %ebx
12: movl %ebx, %ecx
```

Escreva no espaço ao lado o conteúdo dos registos eax, ebx e ecx após a execução da última instrução. Se não for possível determinar o conteúdo de um dos registos escreva ? como conteúdo do registo.

eax 5

ebx 6

ecx 6

**Q5- 1.5 valores** Considere a seguinte sequência de instruções

```
xorl %eax, %eax
incl %eax
movl $0x80000000, %ebx
leal 2(%ebx, %eax, 4), %ecx
    eax * 4 = 0x04    2 = 0x02    soma1 = 0x80000004
```

Escreva no espaço ao lado o conteúdo dos registos eax, ebx, e ecx após a execução da última instrução.

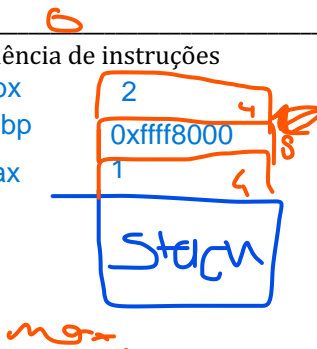
eax 1

ebx 0x80000000

ecx 0x80000006

**Q6- 2.0 valores** Considere a seguinte sequência de instruções

```
movl $0xffff8000, %ebp %ebx
movl $1, %eax %ebp
movl $2 %ebx %eax
pushl %eax
call sub1
cont: add $4, %esp
    . . .
sub1: push %ebp
    movl %esp, %ebp
    push %ebx
    movl -8(%ebp), %eax
    movl $4, %ebx
    addl %ebx, %eax
    movl -4(%ebp), %ebx
    mov %ebp, %esp    %esp = 0xffff8000
    popl %ebp
    ret
```



eax 1

1 1  
1+4 = 5

ebx 2

4  
2

ebp 0xffff8000

Escreva no espaço ao lado o conteúdo dos registos eax, ebx, e ebp depois da execução da instrução máquina que está na etiqueta cont.

**Q7- 2.0 valores** Suponha que se pretende construir o programa prog que é construído a partir de dois ficheiros fonte, um escrito em C ( prog1 . c ) e outro em assembler do Pentium IA-32 ( prog2 . s ). Para construir o programa prog deram-se os seguintes comandos no interpretador de comandos (shell):

as -o prog2.o prog2.s

gcc -o prog prog1.c prog2.o

Explique o que é feito por cada um dos 2 comandos anteriores.

```
extern int myStrlen( char str[])
```

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

```
extern int myStrlen( char str[] );
```

```
int l = myStrlen( s );
```

```
return 0;
```

Foi criado um ficheiro executável main com a seguinte sequência de comandos

```
gcc -o main main.c myStrlen.o
```

Executando o programa main este escreveu no terminal o número 13.

```
int myStrlen( char str[] ) {
```

}

```
.data
    str: .asciiz    "hello, world\n" # reserva de para a cadeia
    # .asciiz significa que a cadeia é terminada por um byte a 0
```

.text

• • •

```
.text
.globl myStrlen
myStrlen:
```

As duas perguntas seguintes destinam-se a ajudar à avaliação dos trabalhos 1 e 2. Se não está a realizar a avaliação laboratorial este ano, por já ter obtido uma nota laboratorial em ano anterior, não deve responder a estas duas perguntas, mas escrever neste espaço “Não estou a realizar a parte laboratorial em 2020/21” e rubricar.

Neste caso, nota final será ajustada de acordo com a fórmula:  $\text{nota}_{\text{Obtida Nas Questões}_1\_a\_7} * 20/16$

### Q9- 2.0 valores

Para o TPC1 foi-lhe proposto que completasse o código de um simulador (escrito na linguagem C) de uma arquitetura composta por um CPU muito simples e uma memória; esta questão é uma extensão do TPC1 e o CPU a simular inclui um novo registo, SP (um *stack pointer*) e duas novas instruções.

O código a completar está no ficheiro `dorun.c` (cuja listagem segue abaixo com as “caixas” para preencher) e deve implementar o ciclo de *fetch*, *decode* e *execute* para simular a execução das novas instruções; apenas para referência (e para ajuda, e para tornar o enunciado mais curto 😊) são apresentadas duas instruções que já existiam no TPC1, HALT e LOAD, e uma terceira, DEC, que simplesmente decrementa o valor em AC.

### Instruction Set Architecture:

O processador usa palavras de 16 bits e tem um único registo geral (AC), um *stack pointer* (SP), e um *Program Counter* (PC), além do registo que guarda a instrução a executar (IR). A memória está também organizada em palavras de 16 bits por endereço, sendo cada endereço de 12 bits.

As instruções têm tamanho fixo de 16 bits, sendo os 4 mais significativos para o código de operação e os restantes 12 para endereço, quando necessário. As instruções suportadas e o respetivo código máquina em representação hexadecimal, são descritas a seguir:

Código	Assembly	Descrição
0x0XXX	HALT	Pára (halt) o CPU
0x1EEE	LOAD EEE	AC <- Mem[EEE]
0x2EEE	DEC	AC <- AC - 1
0x3EEE	CALL EEE	Guarda o endereço da instrução seguinte na pilha e salta para a subrotina em EEE
0x4XXX	RET	Retoma a execução no endereço guardado CALL

XXX significa que os bits são ignorados

EEE representa o endereço a indicar na instrução

Mem[EEE] representa a célula de memória de endereço EEE



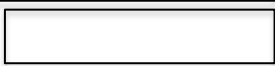
Quando o CPU é ligado (*power-on*), o PC é inicializado com zero (0x000) e o SP com 0xFFFF, para marcar o topo da pilha (*top-of-stack*). A pilha cresce para endereços decrescentes. Em seguida apresenta-se um exemplo de um programa que decrementa um inteiro deixando o resultado no AC:

```
end.    code    assembly
0x000:  0x1010    LOAD 0x010  (inteiro a decrementar)
0x001:  0x3013    CALL 0x003
0x002:  0x0000    HALT
0x003:  0x2000    DEC
0x004:  0x4000    RET
...
0x010:  2          Inteiro (2 como exemplo)
...
0x0FF:  ?          Topo do stack (valor não inicializado)
```

```

extern unsigned short int Mem[];    // A memória está definida num outro módulo C


void dorun(){
    unsigned short int pc, ir, ac, sp;
    unsigned short int opcode;
    unsigned short int address;


    pc = 0;
    while( 1 ) {
        ir =  ; // FETCH uma palavra (16 bits)
        opcode = ir &  ; // opcode, 4 bits mais significativos
        address = ir &  ; // endereço, 12 bits menos significativos

        switch( opcode ){
            // EXECUTE
            case 0x00: /* HALT */
                return;

            case 0x01: /* LOAD */
                ac = Mem[address];
                pc = pc + 1;
                break;

            case 0x02: /* DEC */
                ac = ac - 1;
                pc = pc + 1;
                break;

            case 0x03: /* CALL */
                
                break;

            case 0x04: /* RET */
                
                break;

            default:
                printf("Invalid instruction!\n");
                return;
        }
    }
}

```

## Q10 – 2.0 valores - Sobre o TPC2

Pretende-se nesta pergunta implementar uma função escrita em *Assembly* da arquitetura *Pentium IA-32*, de acordo com as convenções dadas nas aulas, que distribui os utentes de um serviço de saúde por um conjunto de regiões do país às quais pertencem (e.g. Lisboa, Porto, etc.), de modo a avaliar a distribuição demográfica desses utentes. Por simplificação, cada zona é identificada com um número (0, 1, 2, 3, etc.), e o vetor que contém informação sobre os utentes é um vetor de estruturas. Cada estrutura tem dois campos, em concreto dois inteiros -- número de utente e número da região (0, 1, 2,...).

```
#define MAXU 300000          // Número máximo permitido de utentes
#define MAXR 50              // Número de regiões consideradas

typedef struct {
    int numero_utente;        // offset de 0 bytes dentro da estrutura
    unsigned int regioao;     // offset de 4 bytes dentro da estrutura
} utente;

utente utentesServico[MAXU];    // vetor com a descrição dos utentes
unsigned int utentesRegiao[MAXR]; // vetor com o número de utentes por região
```

A função irá ser invocada a partir de um programa em C onde é declarada da seguinte forma:

```
extern void classifica( int utentes[ ], int size, int regioes [ ] );
```

O vetor `utentes` tem `size` posições (e `size` é menor ou igual a `MAXU`) e que `utentes[k].regiao` (com  $k \geq 0$  e  $k < \text{size}$ ) contém apenas valores inteiros entre 0 e `MAXR-1`. O segundo parâmetro da função, vetor `regioes[ ]` tem `MAXR` posições e `regioes[i]` contém o número de ocorrências do valor `i` no campo `regiao` do vetor `utentes`.

O código abaixo exemplifica o uso da função `classifica`.

```
extern void classifica( int utentes[ ], int size, int regioes [ ] );

int main( ) {
    // código que preenche o vetor utentesServico e a variável size
    // omitido
    ...
    for( int i = 0; i < MAXR; i++ )
        utentesRegiao[i] = 0;
    classifica( utentesServico, size, utentesRegiao );
    ...
    // código que processa o vetor utentesRegiao
    // omitido
    return 0;
}
```

Complete o código da subrotina classifica de modo a implementar a funcionalidade descrita:

```
.text
# void classifica( int utentes[ ], int size, int regioes [ ] );

MAXR = 50    # numero de regiões, i.e. dimensão do vetor regioes
SIZEUT = 8   # número de bytes ocupado por cada registo utente
OFFRG = 4    # offset do campo regioao dentro do registo de cada utente
              # o offset do campo numero_utente dentro do registo utente é zero

.globl classifica
classifica:
    push %ebp
    movl %esp, %ebp
    pushl %ebx

    movl [ ], %edx          # endereço do vetor de utentes
    movl $0, %ecx          # contador
    movl [ ], %ebx         # endereço do vetor regioes
11:
    cmpl %ecx, [ ]         # o vetor de utentes chegou ao fim ?
    [ ]                   # se sim, termina subrotina

    mov [ ](%edx), %eax    # consulta o numero da região
    incl [ ]              # incrementa regioes[%eax]

    addl [ ], %edx        # avança para o próximo utente
    [ ]                   # incrementa o contador
    jmp 11

fim:
    [ ]
    [ ]
    [ ]
    ret
```

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