# Teste 1B de Arquitetura de Computadores 18/04/2015 Duração 2h

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

---

**1- 1.5 val** Considere uma representação de inteiros sem sinal usando 8 bits.

**a)** Como é codificado em binário o valor 34 (base 10) ?

00100010

**b)** Qual é o maior valor que se pode representar?

$11111111_2$ ou $255_{10}$

---

**2- 1.5 val** Considere uma representação de inteiros com sinal em complemento para 2 em que estão disponíveis 6 bits.

**a)** Como é representado -5 (base 10) ?

000101 +5
111010 complemento para 1
+1
111011

**b)** Qual é o maior valor que se consegue representar?
Com n bits e sinal o maior valor que se consegue representar é $2^{n-1}-1$ isto é $2^{5}-1$ ou seja 31

**c)** Qual é o menor valor que se pode representar?
Com n bits e sinal o menor valor que se consegue representar é $-2^{n-1}$ isto é $-2^{5}-1$ ou seja -32

---

**3- 2.0 val** Considere um CPU que tem uma unidade aritmética e lógica em que as duas entradas têm 8 bits e a saída tem 8 bits. Os inteiros com sinal são representados em complemento para 2.

**a)** Diga qual é o resultado (em base 2) da soma de $01111111_2$ com $00011011_2$

0111 1111
+ 00011011
10011010

**b)** Indique os valores da CF e OF.

CF = 0
OF=1 porque a carry in do bit 7 é 1 e o carryout é 0

**c)** O que significa a configuração de flags da alínea b) relativamente ao significado do resultado?

Se se trata de uma soma de valores sem sinal o resultado é correcto porque CF=0 isto é o resultado não é maior do que $2^8 - 1$

Se se trata de uma soma de valores com sinal o resultado é incorrecto. A OF está a 1 indicando que houve um overflow. Isto é óbvio porque estou a somar dois valores positivos e o resultado é negativo

**4- 2.5 val** Considere a norma para representação de números reais em precisão simples IEEE 754 em que um número real é representado em 32 bits (bit 31 é o mais significativo e 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: expoente. Sendo X o valor representado, o valor do expoente é X-127
- Bits 22 a 0: mantissa. Sendo a configuração dos bits da mantissa xxxxx...$xxx_2$, o valor efectivo da mantissa é 1.xxxx...$xx_2$

Diga em base 10 qual é o valor representado por 1 10000001 11000000000000000000000. Justifique a resposta.

Sinal 1 número negativo
Expoente 10000001 ou seja 129. Como está em excesso 127 o expoente real é 2
Mantissa é 1.11 (o bit à esquerda do ponto) não é representado
Temos portanto -1.11 x $2^2$ ou seja -$111.0_2$ isto é -$7.0_{10}$

---

**5- 1.5 val** Considere a seguinte sequência de instruções

```
movl   $4,%eax
movl   $6, %ebx
subl   %ebx, %eax
```

Escreva no espaço ao lado o conteúdo dos registos *eax* e *ebx*, bem como os valores das *flags* ZF e SF, após a execução da última instrução da sequência

eax -2

ebx 6

ZF 0

SF 1

---

**6- 1.5 val** Considere a seguinte sequência de instruções

```
        movl  $5, %eax
        cmpl  $6, %eax
        jge   l1
        movl $5, %ebx
        jmp l2
l1:     movl $8, %ebx
l2:     movl %ebx, %ecx
```

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

eax 5

ebx 5

---

**7- 1.5 val** Considere a seguinte sequência de instruções

```
xorl %ecx, %ecx
mov $3, %edx
pushl %edx
pushl %ecx
popl %edx
popl %ecx
```

Escreva no espaço ao lado o conteúdo dos registos *ecx* e *edx*, após a execução da última instrução da sequência.

ecx 3

edx 0

---

**8-** Pretende-se que construa uma função chamada *conta* que tem a assinatura

```
int conta( int *end, int val)
```

- Recebe como parâmetros de entrada o endereço inicial *end* de uma sequência de inteiros e um valor *val.* A sequência de inteiros está terminada por um 0; este 0 final não é considerado pela função;
- Retorna o número de ocorrências de inteiros diferentes de *val* na sequência que se inicia em *end.*

Ver a seguir um exemplo do uso da função:

```
int x[ 6 ]= { 7, 15, 4, 7, 2, 0 };   // declaração e inicialização do array x
...
int a = conta( x, 7 );   // a variável a fica com o valor 3
```

**a)** **2.0 val** Implemente em C a função "conta" descrita acima.

```
int conta( int *end, int val) {
   int *p; int i;
   int cont = 0;   p = end;
   for ( i = 0; *p != 0; i++) {
         if ( *p != val ) cont++;
         p ++;
   }
   return cont;
}
```

**b)** **1.0 val** Assuma que a função "conta" está implementada. Escreva o código assembly correspondente à invocação da função passando o array x e o valor 7 como argumentos:

```
.data
   x:  .int   7, 15, 4, 7, 2, 0
.text
     ...

      pushl $7
      pushl $x
      call  conta
      addl $8,%esp
```

**c)** **2.0 val** Escreva, em *assembly* do Pentium (versão 32 bits) e as mnemónicas e convenções do *gas* (**g**nu **as**sembler) o código da função *conta*.

```
.text
     ...

conta: pushl %ebp
         movl %esp, %ebp
         pushl  %ebx
         movl $0, % eax            # cont (eax) ← 0
         movl  12(%ebp), %ecx      #  ecx ← val
         movl   8(%ebp), %edx      #  edx ← &x[0]
l1:      cmpl   $0, (%edx)         #  *p == 0 ?
         jz      end
         cmpl   (%edx), %ecx       #  *p == val ?
         jnz    l2
         incl  %eax                #  cont++
l2:      addl  $4, %edx            # p++
         jmp  l1
end:     popl  %ebx
         movl  %ebp, %esp
         popl %ebp
         ret
```

**9- 3.0 val** Pretende-se que escreva em *assembly* do Pentium (versão 32 bits) e as mnemónicas e convenções do *gas* (**g**nu **as**sembler) o código equivalente ao seguinte fragmento de código C:

```
#define SIZE 100
typedef struct{
    short int  s;
    char c1;
    char c2;
    float f;
} my_struct;
my_struct ss[SIZE];

int i, cont;

cont = 0;
for( i = 0; i < SIZE; i++)
     if( ss[i].c1 == 'X') cont = cont+1;
```

Sugestão: utilize a forma de calcular o *endereço efetivo* especificado por

*constante( registo_base , registo_índice , factor_de_escala )*

em que o endereço efetivo é dado por *constante + registo_base + registo_índice * factor_de_escala*, em que o factor de escala pode ser 1, 2, 4 ou 8.

```
.data

.comm cont, 4     # reserva de espaço não inicializado para a variável cont
.comm ss, 800     # reserva de espaço não inicializado para o vector de estruturas ss
...
.text
...
# código que inicializa o vector ss e que não interessa para a resolução
...
# código da sua resolução:

          movl  $0, %eax     % cont = 0
          movl  %0, %ecx     % i (ecx) ← 0
          movl  $2, %edx     % distancia de c1 ao início da estrutura
 l1:      cmpl  $100, %ecx   % i == SIZE ?
          jz  fim
          cmpb  $'x', $ss(%edx, %ecx, 8 )  % comparer o carácter x
                                           % com o conteudo da posição
                                           % cujo endereço é o inicio do vector
                                           % mais i * dimensão da estrutura
                                           % mais 2 distancia de c1 ao inicio da estrutura
          jnz  l2
          incl %eax
l2:       incl %ecx
          jmp l1
fim:      movl %eax, cont
```

# 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.
Arguments to instructions: Note that it is not possible for **both** src and dest to be memory addresses.

Constant (decimal or hex): $10 or $0xff
Fixed address: (2000) or (0x1000+53)
Register: %eax %bl
Dynamic address: (%eax) or 16(%esp)
or 200(%edx, %ecx, 4)

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

| Instruction | Effect | Examples |
|---|---|---|
| **Copying Data** | | |
| mov *src,dest* | Copy src to dest | mov $10,%eax<br>movw %eax,(2000) |
| **Arithmetic** | | |
| add *src,dest* | dest = dest + src | add $10, %esi |
| sub *src,dest* | dest = dest – src | sub %eax,%ebx |
| mul *reg* | edx:eax = eax * reg | mul %esi |
| div *reg* | eax = edx:eax / reg<br>edx = edx:eax **mod** reg | div %edi |
| inc *dest* | Increment destination | inc %eax |
| dec *dest* | Decrement destination | dec (0x1000) |
| **Function Calls** | | |
| call *label* | Push eip, transfer control | call format_disk |
| ret | Pop eip and return | ret |
| push *item* | Push item (constant or register) to stack | pushl $32<br>push %eax |
| pop *[reg]* | Pop item from stack; optionally store to register | pop %eax popl |
| **Bitwise Operations** | | |
| and *src,dest* | dest = src & dest | and %ebx, %eax |
| or *src,dest* | dest = src \| dest | orl (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) |
| **Conditionals and Jumps** | | |
| cmp *arg1,arg2* | Compare arg1 to arg2; must immediately precede any of the conditional jump instructions | cmp $0,%eax |
| je *label* | Jump to label if arg1 == arg2 | je endloop |
| jne *label* | Jump to label if arg1 != arg2 | jne loopstart |
| jg *label* | Jump to label if arg2 > arg1 | jg exit |
| jge *label* | Jump to label if arg2 >= arg1 | jge format_disk |
| jl *label* | Jump to label if arg2 < arg1 | jl error |
| jle *label* | Jump to label if arg2 <= arg1 | jle finish |
| jmp *label* | Unconditional relative jump | jmp exit |

## Diretivas (exemplos):

.data – inicio da zona de dados/variávais globais
.int – reserva de memória para variáveis de 32bits
.byte – reserva de memória para bytes (8bits)
.text – inicio da zona de código
.comm *label*, *length* – reserva de length bytes
.ascii – reserva de memória para sequências de chars

## Convenções de chamada de funções:

**Invocador:**
- empilha parâmetros da direita para a esquerda
- call função
- recupera espaço ocupado pelos parâmetros antes empilhados

**Dimensões dos tipos C em ia32:**
char 1 byte
short 2 bytes
int, float, long e *pointer* 4 bytes
double 8 bytes

**Invocado (função):**
- inicia ebp: push %ebp
  mov %esp, %ebp
  sub 4, %esp #se necessário var. local
- usa ebp para endereçar os argumentos, p.e. 4(%ebp)
- resultado fica em %eax (se inteiro ou endereço)
- acaba com: mov %ebp, %esp #liberta var. local
  pop %ebp
  ret