- Consulta apenas das 2 folhas de consulta (C e Assembly).
- A infracção implica, no mínimo, a ANULAÇÃO da prova.
- Quando omissa a arquitectura, considere Linux/IA32.

Versão: A Nota mínima: **7.5/20 valores** / Duração: 120 minutos

Número:_____ **Nome:**_____

Responda aos grupos II, III, IV e V em folhas A4 separadas.

[8v] **Grupo I - Assinale no seguinte grupo se as frases são verdadeiras ou falsas (uma resposta errada desconta 50% de uma correcta).**

	V	F
1) Em C, o valor de um apontador é o endereço do primeiro byte do bloco de memória para o qual aponta	☒	☐
2) Em C, o tipo de dados do apontador determina o espaço em memória necessário para o armazenar	☐	☒
3) Em C, admita a variável "`unsigned char x;`". O maior valor positivo que podemos armazenar em `x` é 2^7 - 1	☐	☒
4) Em C, um *cast* para `char` de uma variável do tipo `unsigned short` com um valor positivo pode resultar num valor negativo	☒	☐
5) Em C, as operações aritméticas com qualquer tipo de dados para valores inteiros seguem as regras da aritmética modular	☒	☐
6) Em C, admita as variáveis "`int x=0xABCD;`" e "`char *ptr=&x`". Logo, "`printf("%hhX",*(ptr+1));`" imprime o valor `0xCD`...	☐	☒
7) Em C, "`x >> 2`" aplica um deslocamento lógico para a direita se `x` for do tipo `unsigned int` e um aritmético se `x` for do tipo `int`	☒	☐
8) Em C, admita o vetor "`short vec[5];`". A função `realloc` permite alterar o tamanho de `vec` para armazenar mais elementos	☐	☒
9) Em Assembly, qualquer que seja o valor armazenado em `%eax`, o resultado de "`sall $4,%eax`" e "`shll $4,%eax`" é o mesmo	☒	☐
10) Em Assembly, depois do prólogo de uma função, o valor antigo de `%ebp` pode ser encontrado em `(%esp)`	☒	☐
11) Em Assembly, reservar 8 bytes para variáveis locais de uma função pode ser conseguido com "`addl $8,%esp`"	☐	☒
12) Em IA32, a *stack* é usada para suportar o retorno do valor de saída de uma função, tal como acontece com o controlo de fluxo	☐	☒
13) Em IA32, a execução da instrução `ret` não altera o valor de qualquer registo	☐	☒
14) Em IA32, o resultado da instrução "`jmp func`" depende do valor dos bits do registo EFLAGS	☐	☒
15) De acordo com a convenção usada em Linux/IA32, uma função pode usar `%edx` sem a necessidade de o salvaguardar e restaurar	☒	☐
16) Admita uma matriz de inteiros alocada na *heap* dentro de uma função. O seu espaço é automaticamente libertado no fim da função	☐	☒
17) As restrições de alinhamento em memória contribuem para a possível fragmentação interna de um bloco reservado na *heap*	☒	☐
18) O tamanho de uma estrutura sujeita a alinhamento é sempre o mesmo em IA32 e x86-64, independentemente dos seus campos	☐	☒
19) O endereço inicial de uma estrutura sujeita a alinhamento depende dos tipos de dados dos seus campos	☒	☐
20) A técnica de otimização de programas que move código para fora de um ciclo é denominada "*loop unrolling*"	☐	☒

[2v] **Grupo II – Responda numa folha A4 separada que deve assinar e entregar no final do exame.**

Foi contratado para implementar um conjunto de funções que operam com um tipo de dados que agrupa **4 bytes com sinal num único valor de 32 bits sem sinal**:

```
typedef unsigned int packed_t;
```

O seu antecessor (que foi despedido por incompetência) produziu a função ao lado para extrair o byte indicado e expandi-lo para um valor de 32 bits com sinal. Os bytes são numerados de 0 (menos significativo) a 3 (mais significativo).

```
/* Extract byte from word. Return as signed
integer */

int xbyte(packed_t word, char byte_num){
   return (word >> (byte_num << 3)) & 0xFF;
}
```

[1v] **a)** Qual o problema da função desenvolvida? **Justifique a sua resposta.**

Esta questão realça a diferença entre realizar a extensão do sinal de um número por comparação à adição de zeros à esquerda quando se aplica um deslocamento. A função apresentada não faz qualquer extensão de sinal. Por exemplo, se tentarmos extrair o byte zero do valor 0xFF obtemos o valor 255 e não -1, como seria suposto.

[1v] **b)** Escreva uma versão correta da função. **Comente o seu código, descrevendo a abordagem seguida.**

Primeiro, aplica-se um deslocamento para a esquerda de forma a colocar o bit mais significativo do byte que pretendemos extrair na posição 31. Depois, aplica-se um deslocamento para a direita de 24 bits, movendo o byte a extrair para a posição adequada (byte menos significativo), aplicando ao mesmo tempo a necessária extensão de sinal.

```
int xbyte(packed_t word, char byte_num){
   int left = word << ((3-byte_num) << 3);
   return left >> 24;
}
```

[5v] **Grupo III – Responda numa folha A4 separada que deve assinar e entregar no final do exame.**

Considere as seguintes declarações:

```
typedef struct {
  char a;
  char b[2];
  int c;
  unsigned short d;
  structB *e;
  char f;
}structA;
```

```
typedef struct {
  int a;
  char b;
  short c;
  long int d;
}structB;
```

```
typedef union{
  int a;
  int b;
  int c[2];
  unsigned char d[8];
}unionC;
```

[1.5v] **a)** Indique o alinhamento dos campos de uma estrutura do tipo `structA`. Indique claramente, para cada campo, o seu endereço, bem como as partes alocadas mas não usadas para satisfazer as restrições de alinhamento. Indique o tamanho total da estrutura. **Admita que a estrutura está colocada a partir do endereço `0x100`.**

```
typedef struct {
  char a;                  0x100: 1 byte
  char b[2];               0x101: 2 bytes
  [gap]                    0x103: 1 byte
  int c;                   0x104: 4 bytes
  unsigned short d;        0x108: 2 bytes
  [gap]                    0x10A: 2 bytes
  structB *e;              0x10C: 4 bytes
  char f;                  0x110: 1 byte
  [gap]                    0x111: 3 bytes
}structA;
```

Tamanho da estrutura: 20 bytes

[1.5v] **b)** Considerando o seguinte fragmento de código em C, que valores irão ser impressos? **Justifique a sua resposta.**

```
  unionC u;
  u.a = 0x01020304;
  u.b = 0x05060708;

  u.d[4] = 0x0A;   u.d[5] = 0x0B;
  u.d[6] = 0x0C;   u.d[7] = 0x0D;
```

```
printf("%d\n",sizeof(u));
printf("%X\n",u.c[0]);
printf("%X\n",u.c[1]);
```

8 → O tamanho da *union* é igual ao maior dos seus campos.

0x05060708 → u.b e u.c[0] são duas referências para os mesmos 4 bytes (os menos significativos da *union*).

0x0D0C0B0A → u.c[1] e u.d[4] a u.d[7] são referências para os mesmos 4 bytes (os mais significativos da *union*). Como a arquitectura IA32 é *little endian*, ao serem interpretados como um único valor inteiro pela função *printf*, o byte menos significativo desse inteiro é o byte com o endereço menor. Logo, o inteiro resultante tem os bytes pela ordem indicada.

[2v] **c)** Considere o seguinte fragmento de código em C:

```
structA matrix[4][5];

void fill_structB_b(int i, int j){
   matrix[i][j].b[0] = matrix[i][j].a;
   matrix[i][j].b[1] = matrix[i][j].f;
}
```

Reescreva a função `fill_structB_b` em Assembly. Na sua resolução tenha em consideração que `matrix` é uma matriz de estruturas global definida estaticamente. **Comente o seu código.**

```
fill_structA_b:
       pushl %ebp
       movl %esp, %ebp
       movl 8(%ebp), %eax              # i
       imull $5,%eax                   # i*5 (nº de elementos por linha)
       imull $20, %eax                 # i*5*20 (tamanho da estrutura)
       movl 12(%ebp), %edx             # j
       imull $20, %edx                 # j*20 (tamanho da estrutura)
       leal matrix(%eax,%edx), %eax    # &matrix[i][j]
       movb (%eax),%cl                 # %cl = matrix[i][j].a
       movb %cl,1(%eax)                # matrix[i][j].b[0] = %cl
       movb 16(%eax), %cl              # %cl = matrix[i][j].f
       movb %cl, 2(%eax)               # matrix[i][j].b[1] = %cl
       movl %ebp, %esp
       popl %ebp
       ret
```

[3v] **Grupo IV – Responda numa folha A4 separada que deve assinar e entregar no final do exame.**

[1v] **a)** Mostre como os seguintes valores são armazenados em memória em IA32. Apenas preencha os bytes que se aplicam (isto é, se um valor não usar todos os bytes, assegure-se que não preenche nada nos bytes não ocupados). Assuma que os valores têm como endereço 0x100.

Valor	0x100	0x101	0x102	0x103
"ABC"	'A'	'B'	'C'	'\0'
0xABCD	CD	AB		

[1v] **b)** Use os seguintes valores iniciais em memória e nos registos da arquitetura para responder a cada uma das questões (isto é, cada questão não é afetada pela execução das instruções anteriores).

Endereço	Valor
0x100	0xFF
0x104	0xAB
0x108	0x13
0x10C	0x11

Registo	Valor
%eax	0x100
%ecx	0x1
%edx	0x3
%ebx	0x4

A. Qual o novo valor de `%eax` após "`movl 0x100,%eax`"? __**0xFF**__

B. Qual o novo valor de `%ecx` após "`movl (%eax,%edx,4),%ecx`"? __**0x11**__

C. Qual o endereço que é alterado com "`subl %edx,4(%eax)`"? __**0x104**__

D. Qual o novo valor de `%ebx` após "`leal 0x100(,%ebx,2),%ebx`"? __**0x108**__

[1v] **c)** Converta a instrução "`leal 0x4(%eax,%ecx,8),%ebx`" num conjunto equivalente de instruções Assembly. **Assegure-se de que o código convertido e a instrução `leal` original deixam os registos `%eax`, `%ecx` e `%ebx` com os mesmos valores.** Na sua solução não pode usar outros registos para além destes 3.

```
movl %ecx, %ebx
imull $8, %ebx
addl %eax, %ebx
addl $4, %ebx
```

```
# As duas primeiras linhas podiam ser substituídas por:
movl $8, %ebx
imull %ecx, %ebx

# As duas últimas linhas podiam ser substituídas por:
addl $4, %ebx
addl %eax, %ebx
```

[2v] **Grupo V – Responda numa folha A4 separada que deve assinar e entregar no final do exame.**

Um grupo de alunos do DEI decidiu participar num concurso de programação e há ainda uma vaga na equipa. Para ser admitido terá de desenvolver a versão mais otimizada do cálculo do fatorial de um número. A sua primeira abordagem foi reescrever a versão recursiva de forma iterativa, obtendo a função descrita em `fact_iter`:

```
int fact_iter(int n){
 int i, result = 1;

 for(i = n; i > 1; i--){
   result = result * i;
 }
 return result;
}
```

```
int fact_unroll2(int n){
 int i, result = 1;

 for(i = n; i > 0; i-=2){
  result = (result*i)*(i-1);
 }
 return result;
}
```

Ciclos de relógio Por Elem/Op (CPE)

Operação (inteiros)	Latência Operação	Débito Operações
Adição	1	1
Multiplicação	4	1
Divisão	36	36

[1v] **a)** Ao testar a performance da versão `fact_iter` num processador superescalar com as características da tabela acima verificou que reduziu o número de CPE de 64, obtidos pela versão recursiva, para cerca de 4. Explique o aumento de performance e o valor encontrado.

A eliminação do *overhead* inerente às sucessivas invocações da função (e consequente retorno) na versão recursiva justifica o aumento de performance verificado. O valor de 4 CPE justifica-se pela latência da operação de multiplicação, impondo um limite da performance desta versão.

[1v] **b)** A seguir, decidiu aplicar a técnica de "*loop unrolling*" para continuar a melhorar a performance da função num processador superescalar e chegou à versão descrita em `fact_unroll2`. No entanto, ficou desapontado quando verificou que a performance não é melhor do que a versão `fact_iter`. Um colega sugeriu-lhe que alterasse a linha dentro do ciclo para "`result = result * (i * (i-1));`" e a performance da sua função passou para um CPE de 2.5. Como explica esta melhoria na performance?

A alteração quebrou a interdependência da operação de multiplicação entre os dois valores processados em cada iteração do ciclo. Na nova versão, a multiplicação `i*(i-1)` pode ocorrer em simultâneo com a multiplicação pelo valor da variável `result` determinado na iteração anterior.