Representação de informação

A1
Resolva os seguintes problemas.

a) Mostrando os cálculos que efectuar, converta para hexadecimal o valor $0.26075*10^3$

Sugestão de resolução

Na conversão de um número em decimal para hexadecimal, é possível optar por uma de 2 alternativas (ambas correctas):
- divisões/multiplicações sucessivas por 16 (método directo), ou
- converter primeiro para binário (evitando, se possível, a realização de divisões/multiplicações por 2) e depois, sem operações, associar os bits em grupos de 4, para a esquerda/direita do ponto decimal, e atribuir a cada grupo de 4 bits o correspondente valor hexadecimal.

Este pb tem, contudo, uma mui ligeira complexidade adicional: o valor fornecido na base 10 é apresentado, não sob a forma de um nº contendo uma parte inteira e uma parte fraccionária (como se viu nas sessões teórico-práticas), mas sim com a chamada notação científica (uma parte fraccionária que multiplica por uma potência de 10).
Quando confrontados com uma questão deste tipo, certos estudantes ficam confusos e, por vezes, atacam logo com a resolução do pb sem pensar bem no assunto, i.e., assumem que este valor é constituído por 2 partes
- a parte fraccionária 0.26075, e
- o expoente 3,

convertem cada um desses números em separado para a nova base, e apresentam o resultado como sendo o produto da parte fraccionária (na nova base) pela nova base elevada ao expoente (convertido para a nova base). **ISTO É UM DISPARATE!**
Podem comprová-lo experimentando fazer uma conversão de um nº pequeno e simples - como por ex., o valor 5 escrito na forma $0.5*10^1$ - e analisando criticamente o resultado dessa conversão:
- pelo método disparatado daria 0.1_2*2^1; como a multiplicação de um nº pela sua base corresponde a deslocar o ponto decimal 1 casa para a direita, o resultado final desta multiplicação daria o valor 1_2. Acha que 5 no sistema decimal é igual a 1 no sistema binário?

Assim, antes de fazermos qq operação de conversão de bases, temos primeiro de passar este valor da notação científica para a outra notação; neste caso pegar no valor fornecido, fazer a multiplicação lá especificada pela potência da base 10, para se obter o valor 260.75. E é este o valor que irá ser convertido para hexadecimal.

Neste caso concreto (e na maioria dos casos) é mais simples passar primeiro para binário e depois para hexadecimal.

Se ainda não resolveu este pb, tente agora fazê-lo antes de espreitar pelo resto da resolução...

Vamos então à conversão, por partes:

- parte inteira: 260 ; vamos evitar as divisões sucessivas, e tentar encontrar as potências de 2 que, somadas, perfazem esse valor; sabendo a tabuada de cor, recordamos que a potência de 2 mais próxima de 260 e <260 é 256 ($=2^8$); para 260 faltam 4, que é outra potência de 2 ($=2^2$); então $260 = 2^8 + 2^2$, ou seja, $260=100000100_2$;

- parte fraccionária: 0.75 ; vamos evitar as multiplicações sucessivas, e tentar encontrar as potências negativas de 2 que, somadas, perfazem esse valor; este caso é demasiado simples: $0.75=0.5+0.25 = 1/2 + 1/2^2$, ou seja $0.75=0.11_2$;

Temos então que 260.75 = 100000100.11_2 .
Para passar para hexadecimal é só agrupar em conjuntos de 4 bits, a contar do ponto decimal, para a esquerda e direita, acrescentando os zeros que forem precisos e que não alteram o valor: $0001\ 0000\ 0100.1100_2$.
Então, 260.75 = 0x104.C ou $104.C_{16}$.

b) Um registo de 16 bits contém o conjunto de bits correspondente ao valor 0xD8. Mostre o valor que está lá guardado, em decimal, conforme o registo contém um valor inteiro sem sinal, ou o registo contém um valor inteiro (codificado em complemento para 2). Justifique os resultados apresentados.

Sugestão de resolução

O valor 0xD8 em binário é 1101 1000 1101 1000 $_2$.

Alguns pensarão que, como o bit mais à esquerda é 1, este valor em complemento para 2 representa um valor negativo. Será?

Pense e responda antes de continuar!

Atenção ao enunciado: esse valor está num registo de 16 bits; logo o que lá se encontra é o valor binário $0000\ 0000\ 1101\ 1000_2$.

E agora, para codificar em complemento para 2, o que se faz? Como é "complemento para 2", temos de trocar todos os bits e somar 1, verdade?

Pense e responda antes de continuar!

DISPARATE!!

Este valor em complemento para 2 representa um valor positivo, que vale em decimal tanto quanto um valor sem sinal!

Portanto, quer o registo contenha um valor inteiro sem sinal, ou um valor inteiro representado em complemento para 2, o valor em decimal é o mesmo e vem dado por $2^7+2^6+2^4+2^3 = 216$.

c) Efectuando o menor nº possível de operações, represente na base 3 os seguintes valores: 11, 33, 99
*Nota (para **R**): veja se consegue com apenas 2 divisões, explicitando o raciocínio seguido.*

Sugestão de resolução

Comecemos pelo 1º nº, o 11; na 1ª divisão pela nova base (3) daria quociente 3 e resto 2 (que será o 1º algarismo à esquerda do ponto decimal, no sistema de numeração de base 3); dividindo agora o quociente (3) novamente pela base (3) teríamos 0 de resto (o 2º algarismo à esq do ponto decimal) e 1 de quociente (o 3º e último algarismo).

Mas, será que precisávamos de fazer estas 2 divisões por 3? Neste caso tão simples, não dá para ver que $11 = 3^2+2$, ou seja, $11 = 1*3^2+2*3^0$?
Assim, neste caso, $11 = 102_3$.

E agora 33: não será fácil de constatar que $33=11*3$, e que $99=11*9=11*3^2$?

Se ainda não resolveu este pb, tente agora fazê-lo antes de espreitar pela resolução...

Ainda não conseguiu?
E não se lembra do que leu na resolução de a) ? Que a multiplicação de um nº pela sua base correspondia a deslocar o ponto decimal de uma casa para a direita? E no caso de um inteiro (sem ponto decimal, embora se possa subentendê-lo onde deveria estar), multiplicar pela sua base corresponde a deslocar todo o nº uma casa para a esquerda, acrescentando um zero à direita do nº.
Tente outra vez...

Resolução:

$33 = 11*3 = (1*3^2+2*3^0)*3^1 = 1*3^3+2*3^1 = 1020_3$

$99 = 11*3^2 = (1*3^2+2*3^0)*3^2 = 1*3^4+2*3^2 = 10200_3$ ou
$99 = 33*3 = (1*3^3+2*3^1)*3^1 = 1*3^4+2*3^2 = 10200_3$

d) Uma variável em C declarada como `unsigned short int` *(16 bits) é inicializada com um valor que corresponde à dimensão da memória cache no início dos processadores IA-32 com cache interna (i.e., 40K). Mostre, em hexadecimal, qual o valor que é guardado em memória.*
*Notas (para **R**): (i) tente resolver o problema sem efectuar qualquer divisão aritmética, mostrando os passos que seguir; e resolva adicionalmente o seguinte: (ii) repita o exercício para o dobro desse valor (80K), e (iii) repita o exercício mas considerando agora que a variável foi declarada como* `float`*, e efectuando o menor nº possível de operações..*

Sugestão de resolução:

Recorde a tabuada das potências de 2... O nº 40 não lhe diz nada?
Se não, então calcule 40*1024 e depois faça as sucessivas divisões por 2, aproveitando os restos das divisões.

Se prefere antes usar a cabeça, então...
$40K = (32+8)*1K = (2^5+2^3)*2^{10} = 2^{15}+2^{13} = 1010\ 0000\ 0000\ 0000_2 = 0xA000$.

Para representar o dobro desse valor, isso corresponderia a multiplicar pela base (2), ou seja, deslocar o nº uma casa para a esquerda, acrescentando um zero à direita; mas, como em binário com 16 bits (sem sinal) o bit mais à esquerda neste caso é diferente de zero, se o deslocarmos um bit à esquerda vamos perdê-lo.
Esta variável, da maneira como foi declarada em C, não pode usar mais de 16 bits.
Logo, não é possível representar 80K com 16 bits, pois ultrapassa a capacidade máxima de representação de nºs com 16 bits (máx 2^{16}-1, ou seja, 64K-1).

Mas... se se pretende representá-lo em vírgula flutuante, então temos que preparar a expressão que nos calcula os 80K e adaptá-la à expressão que nos dá o valor em decimal de um nº em vírgula flutuante, precisão simples, no IEEE 754:

$V = (-1)^S * 1.F * 2^{E-127} =$

$V = 2 * 40K = 2 * (32+8) * 1K = 2 * (2^5+2^3) * 2^{10} = (2^5+2^3) * 2^{11} = 10\ 1000_2 * 2^{11} = 1.01_2 * 2^{16} = (-1)^0 * 1.01_2 * 2^{E-127}$

Daqui se tira que:
$S = 0$
$F = 010..0_2$ (23 bits no total)
$E = 127+16 = 143 = 2^7+2^3+2^2+2^1+2^0 = 1000\ 1111$ 1000 1111 $_2$

Donde, em hexadecimal, o valor que será guardado em memória virá dado por (os bits do expoente estão sublinhados)
$0\underline{100\ 0111\ 1}010\ 0000\ 0000\ 0000\ 0000_2 = 0x47A00000$

A2

Um "cardeal" de LEI tentava explicar a um colega seu como é que estava organizada a página desta disciplina na Web.
Analise, com espírito crítico e usando os seus conhecimentos, a afirmação do "cardeal", conforme anotada no caderno do seu colega:
"A página da disciplina SC é um ficheiro HTML com logotipos, textos formatados, datas e ...
Tudo isto está guardado num único ficheiro num PC:
- os textos são caracteres alfanuméricos codificados em ASCII 7-bits;
- a formatação dos textos são comandos específicos que estão codificados em binário conforme o instruction set do Pentium;
- os logotipos são imagens JPEG;
- as datas, como são valores inteiros, estão codificados e guardadas no PC em complemento para 2"

Sugestão de resolução

Vamos analisar o pb, relembrar alguns conceitos para compreender o enunciado, e reflectir sobre a sua resolução; assim:

- afirma-se que a pág na Web é um ficheiro HTML, o que parece fazer sentido, uma vez que o HTML é uma linguagem de _____ de _____; logo, um ficheiro HTML é um ficheiro puro de _____, e portanto apenas contém dados codificados de uma única maneira, i.e., codificados em _____ ; *(não avance enquanto não compreender/resolver cada um destes pontos)*
- se toda a informação no ficheiro é puro _____ e está codificada em _____ , então parece haver alguns disparates no meio desta afirmação...
- nomeadamente, não é possível armazenar neste único ficheiro de _____ informação codificada de várias maneiras, como muito provavelmente acontecerá com imagens... logo a info toda não poderá estar num único ficheiro, mas sim em ficheiros separados;
- se os caracteres alfanuméricos estão codificados em ASCII 7-bits, haverá bits que cheguem para codificar os caracteres acentuados (minúsculas e maiúsculas) e as cedilhas?...
- e que tem a ver "formatação de textos" com *instruction set* do Pentium? O 1º é constituido por comandos, mas comandos para a aplicação que vai interpretar o ficheiro HTML (o *browser*), e não são mais que _____ misturado com o restante _____ ; quanto ao 2º são um conjunto de comandos sim, mas são os comandos que o Pentium executa (formatados em binário de forma compactada conforme especificação da Intel para os processadores de 32 bits, e conhecido tb por IA-32) sempre que o computador/processador está a correr um qualquer programa;
- os logotipos são imagens, e quando se fala em imagens associa-se logo o formato JPEG; mas, será este formato o mais adequado para representar desenhos com contornos bem definidos e com um conjunto muito reduzido de cores distintas?...
- e as datas... é verdade que são valores inteiros, mas... as datas num documento de texto estão lá como caracteres alfanuméricos (e como tal codificados) e não como números sobre os quais se vão efectuar operações aritméticas! Se estivéssemos a falar de uma folha de cálculo com valores numéricos para sobre eles se efectuarem operações, então já poderíamos pensar numa representação distinta da alfanumérica.

Julgo que com esta informação já não é preciso dizer mais nada. Falta apenas construir a resposta em Português simples e claro, eventualmente reutilizando algum deste texto..

A3
A gama de valores representáveis (na base decimal) num registo do Pentium está no intervalo:

a) `[-2^31, +2^31[`, sempre que o registo contiver o valor de uma variável numérica que foi declarada num programa em C; comente esta afirmação.

*Nota: a questão colocada nesta alínea tem uma subtileza para uma classificação **R**, ou mesmo **B**. Para uma classificação de apenas **A**, leia a sua reformulação na resolução (sugestão aos estudantes que pretendem mais que **A**: tentem encontrar essa "subtileza" antes de saltarem para a resolução).*

Sugestão de resolução

Se não descobriu a "subtileza", eis a questão reformulada:

a) `[-2^31, +2^31[`, sempre que o registo contiver o valor de uma variável numérica que foi declarada como `int` num programa em C; comente esta afirmação.

O Pentium é o processador mais actual da família IA-32; como tal, as variáveis do tipo `int` são representadas em complemento para 2, com 32 bits. A gama de valores inteiros (negativos e positivos) representáveis em binário, com n bits, é de 2^n; em complemento para 2, há apenas uma representação do zero, sendo possível representar mais um valor inteiro negativo que positivo.
Assim, o intervalo de representação de valores será assimétrico, sendo possível representar 2^{n-1} valores negativos e apenas 2^{n-1}-1 valores positivos, para além do zero.

Então, com 32 bits, a gama de valores representáveis no Pentium é `[-2^31, +2^31[`, conforme indicado no enunciado.

Voltando agora à questão original.
Na questão inicial não se especificava a maneira como a variável numérica tinha sido especificada: poderia ter sido como `int`, `short int`, `unsigned int`, `unsigned short int`, ou até como `float` ou `double` .
De todas estas formas de especificar, apenas uma especificação do tipo `int` poderia produzir o resultado indicado no enunciado; em todos os outros casos, o intervalo de valores seria distinto.

Sugestão adicional aos pretendentes de uma boa classificação: indique os intervalos para os 5 restantes tipos acima referidos. O resultado está no fim da alínea seguinte.

A gama de valores representáveis (na base decimal) num registo do Pentium está no intervalo:

a) (...)
b) `[0, +2^32]` se o registo for o IP/PC; comente esta afirmação.

Sugestão de resolução

O registo *Instruction Pointer* (tradicionalmente conhecido como *Program Counter*) contém o apontador para a localização em memória da próxima instrução (em linguagem máquina) a ser executada pelo processador.

Sendo um endereço de memória, representa um valor inteiro sem sinal.
A gama de valores representáveis em binário, com n bits, é de 2^n; vai de zero a 2^n-1, inclusivé.

Então, com 32 bits, a gama de endereços de memória representáveis em registo num Pentium é `[0, +2^32[`, e não o intervalo indicado no enunciado.

E agora a resolução à questão adicional colocada na sugestão de resolução da alínea anterior:

- `short int, [-2^15, +2^15[`
- `unsigned int, [-0, +2^32[`
- `unsigned short int, [0, +2^16[`
- `float, ]-2^128, +2^128[`
- `double, ]-2^1024, +2^1024[`.

A gama de valores representáveis (na base decimal) num registo do Pentium está no intervalo:

a) (...)
b) (...)
c) [0, 9999] se o valor estiver representado como uma string de caracteres alfanuméricos em ASCII; comente esta afirmação.

Sugestão de resolução

Se pretendemos representar um nº como uma *string* de caracteres alfanuméricos em ASCII, então cada algarismo será codificado como um caracter ASCII, não necessitando de ocupar mais que uma célula de memória (8-bits).
Como cada registo do Pentium tem 32 bits, é possível escrever num registo todo o conteúdo de uma *string* de 4 elementos, i.e., é possível guardar num registo um nº codificado com 4 caracteres alfanuméricos, e como tal estando no intervalo '0000' a '9999'.

Assim, o intervalo de valores especificado no enunciado está de acordo com este raciocínio.

A gama de valores representáveis (na base decimal) num registo do Pentium está no intervalo:

a) (...)
b) (...)
c) (...)
d) `[-2^128, +2^128]` se fôr o conteúdo de um registo de vírgula flutuante de 32 bits; comente esta afirmação.
*Nota: a resolução correcta desta alínea apenas é exigida para a classificação **R**.*

Sugestão de resolução

A representação de valores em vírgula flutuante no Pentium segue a norma IEEE 754; se se usarem apenas 32 bits, então segue a parte da norma referente à precisão simples.
A norma contempla a representação de valores normalizados, valores não-normalizados (os muito próximos de zero), e alguns casos especiais (incluindo o zero e os infinitamente grandes, quer positivos quer negativos).
Como se pretende a gama de valores representáveis, desde o muito grande negativo, até ao muito grande positivo, esta gama é integralmente definida apenas pelos valores normalizados.

E o valor decimal de um nº representado em binário normalizado, segundo o IEEE 754 vem dado por

$$V = (-1)^S * 1.F * 2^{E-127}$$

Então pretende-se o maior valor de F e de E para os valores de S=0 e S=1; isto corresponde a F=1..1_2 (23 bits) e E=1111 1110_2 (para ser normalizado tem de ser diferente de tudo uns), i.e., E=254.

A gama de valores encontra-se então entre $-1.11111111111111111111111_2*2^{127}$ e $+1.11111111111111111111111_2*2^{127}$, incluindo estes 2 valores extremos.
Os valores logo a seguir a estes, mas já fora do intervalo de valores representáveis, seriam $-2*2^{127}$ e $+2*2^{127}$, o que corresponde a representarmos a gama de valores como estando no intervalo `]-2^128, +2^128[`, e não no intervalo indicado no enunciado.

R1

Durante a execução do seguinte fragmento de código C, compilado para um processador compatível com IA-32, o espaço de memória atribuído às variáveis locais `fval,ival,sval,tamp` *estende-se contiguamente, no sentido dos endereços crescentes, com início em* `0xBFFFDA0` *.*
A sequência de valores `00 50 89 43` *, expresso em formato hexadecimal, refere-se ao conteúdo dessas primeiras quatro células de memória.*

```
char tamp[3]="AJP"; // 'A' = 65 em decimal //
short int sval;
int ival;
float fval;
.....
.....
fval = ??? // valor especificado no enunciado //
ival = (int) fval;
sval = (short int) -fval; // notar o sinal menos //
```

a) Apresente o valor em decimal atribuído a cada uma das variáveis `fval, ival, sval.`

Sugestão de resolução

De acordo com o enunciado, a variável `fval` é uma variável declarada como sendo de vírgula flutuante, de precisão simples (32 bits, norma IEEE 754, como acontece em computadores compatíveis com o IA-32); e essa variável está em memória, ocupando as 4 células (32 bits = 4*8) que se seguem a `0xBFFFDA0` . O conteúdo dessas 4 células é tb indicado no enunciado: "`00 50 89 43` *, expresso em formato hexadecimal*"; se nos lembrarmos que o IA-32 é *little endian*, sabemos que na 1ª célula se encontra o *byte* menos significativo, e na última estará o mais significativo.

Então, o conteúdo da variável `fval` representado em hexadecimal e binário, de acordo com a norma IEEE, é, respectivamente (os bits do expoente estão sublinhados),
0x43895000 e 0<u>100 0011 1</u>000 1001 0101 0000 0000 0000_2

S = 0 (valor positivo)
E = 1000 0111 1000 0111 $_2$ => normalizado, logo em notação excesso 127 ; subtraindo 127 para se saber quanto vale o expoente: subtrai-se 128 (remove-se o 1 à esquerda) e soma-se 1; dá 1000_2 = 8
F = 0.0001 0010 101_2

Sabendo que o valor em decimal de um nº em vírgula flutuante, normalizado, precisão simples, representado pela norma IEEE 754, é calculado pela expressão
$V = (-1)^S * 1.F * 2^{E-127}$
então
fval = $(-1)^0$ * 1.0001 0010 101_2 * 2^8 = 1 0001 0010 1 0001 0010 .101_2 = 256+16+2+1/2+1/8 **= 274.625**

De acordo com o fragmento de código C disponibilizado, `ival` é o resultado da conversão de um valor em vírgula flutuante (`fval`) para um valor inteiro (codificado em complemento para 2), através da truncatura da parte decimal; neste caso concreto,
ival = `(int)` 274.625 **= 274**

Também de acordo com o fragmento de código C disponibilizado, `sval` é o resultado da conversão de um valor em vírgula flutuante (`-fval`) para um valor inteiro (codificado em complemento para 2), "curto" (16 bits), através da truncatura da parte decimal; neste caso concreto,
sval = `(short int)` −274.625 **= −274**

Durante a execução do sequinte fragmento de código C, compilado para um processador compatível com IA-32, o espaço de memória atribuído às variáveis locais `fval,ival,sval,tamp` *estende-se contiguamente, no sentido dos endereços crescentes, com início em* `0xBFFFDA0`.
A sequência de valores `00 50 89 43`, *expresso em formato hexadecimal, refere-se ao conteúdo dessas primeiras quatro células de memória.*

```
char tamp[3]="AJP"; // 'A' = 65 em decimal //
short int sval;
int ival;
float fval;
.....
.....
fval = ??? // valor especificado no enunciado //
ival = (int) fval;
sval = (short int) -fval; // notar o sinal menos //
```

a) (...)
b) Considerando que o valor ASCII do caracter ´A´ é 65 (decimal), represente, em hexadecimal, no espaço de memória reservado, a variável `tamp`.

Sugestão de resolução

A tabela ASCII codifica as letras do alfabeto com códigos adjacentes e crescendo com a ordem alfabética; assim, se 'A' é codificado por 65, 'J' deverá ser codificado por 74, e 'P' por 80.
Por outro lado, numa *string* (que é representada normalmente por um vector de caracteres), o conjunto de caracteres que lhe são assim atribuídos são colocados nos primeiros elementos do vector, e se esses não encherem a totalidade do vector, então são atribuídos caracteres *null* (ASCII '0', ou como normalmente se designa em C, '\0') aos restantes elementos do vector (*string*).
Assim, o vector ***tamp[3]***, que tem reservado na memória espaço para 4 elementos de 1 *byte*, contém os seguintes valores em hexadecimal (resultantes da codificação directa dos valores em decimal acima referidos, 65, 74, 80 e 0): **41 4A 50 00**.

Durante a execução do sequinte fragmento de código C, compilado para um processador compatível com IA-32, o espaço de memória atribuído às variáveis locais `fval,ival,sval,tamp` *estende-se contiguamente, no sentido dos endereços crescentes, com início em* `0xBFFFDA0`.
A sequência de valores `00 50 89 43`, *expresso em formato hexadecimal, refere-se ao conteúdo dessas primeiras quatro células de memória.*

```
char tamp[3]="AJP"; // 'A' = 65 em decimal //
short int sval;
int ival;
float fval;
.....
.....
fval = ??? // valor especificado no enunciado //
ival = (int) fval;
sval = (short int) -fval; // notar o sinal menos //
```

a) (...)
b) (...)
c) Represente, em hexadecimal, todas as variáveis declaradas, no fragmento de código, com a indicação precisa dos endereços ocupados na memória.

Sugestão de resolução

São 4 as variáveis declaradas neste fragmento de código C: `fval,ival,sval,tamp`.
O valor em decimal das primeiras 3 variáveis foi calculado em a); falta agora recuperar/calcular os seus valores em hexadecimal:
- `fval` está no texto do enunciado;
- `ival` é a representação em binário/hexadecimal, complemento para 2, de '274': 0000 0000 0000 0000 0000 0001 0001 0010_2 <=> 0x112
- `sval` é a representação em binário/hexadecimal, complemento para 2, com 16 bits, de '−274': 1111 1110 1110 1110_2 <=> 0xFEEE

O valor em hexadecimal da 4ª variável foi calculado na alínea anterior.

Localização das variáveis: `fval` (ocupando 4 células) em `0xBFFFDA0` ; `ival` (4 células) em `0xBFFFDA4` ; `sval` (2 células) em `0xBFFFDA8` ; `tamp` (4 células) em `0xBFFFDAA` .

Resumindo, sem esquecer que o IA-32 é *little endian*:

```
0x 0B FF FD A0 :00
       "    A1 :50
       "    A2 :89
       "    A3 :43
0x 0B FF FD A4 :12
       "    A5 :01
       "    A6 :00
       "    A7 :00
0x 0B FF FD A8 :EE
       "    A9 :FE
0x 0B FF FD AA :41
       "    AB :4A
       "    AC :50
       "    AD :00
```

E se estivéssemos a analisar um computador *big endian*, em vez de um baseado no Pentium?

A diferença está apenas na maneira como uma quantidade escalar, não estruturada - e cuja dimensão em nº de bits é superior à de uma célula de memória - é armazenada nessas células. E o pb do *big* versus *little endian* apenas se coloca neste caso.
No nosso exercício, temos 3 variáveis escalares e uma estruturada (uma *string*, representada como um *array*), cujos elementos são caracteres (e portanto requerendo cada apenas 8 bits, a dimensão de uma célula). Apenas as variáveis escalares irão ser afectadas por esta questão. Então, aqueles 14 *bytes* ficariam assim armazenados em memória, a partir do endereço inicial

```
0x 0B FF FD A0 : 43 89 50 00 00 00 01 12 FE EE 41 4A 50 00
```

R2

Numa aula de laboratório de SC, os grupos estavam a analisar o comportamento do processador na execução de um programa em C; mais concretamente, estavam a visualizar os conteúdos dos registos (em hexadecimal) após terem interrompido a execução a meio.
De repente, um grupo comenta: "Olha! Que coincidência! Os registos `%esp` *(stack pointer) e* `%eax` *(um outro que foi usado para representar uma das variáveis inteiras do programa) têm o mesmo valor, e portanto representam o mesmo valor decimal!"*
E comenta um outro grupo: "E connosco também acontece o mesmo!!"
O docente vai ver ambos os terminais e confirma que os valores visualizados em cada monitor são o mesmo, embora sejam diferentes de um grupo para o outro. E comenta: "É verdade que esses 2 registos têm o mesmo conteúdo em binário! Mas um dos grupos fez uma afirmação correcta, e o outro não!"

a) Encontre uma explicação válida para a posição do docente.
Nota: se esta questão não contivesse a alínea b), seria para uma classificação ***B***

Sugestão de resolução

Vamos analisar cuidadosamente a afirmação do docente: (i) ele confirma a veracidade dos dados visualizados no monitor, i.e., que os registos `%esp` e %eax têm a mesma representação em hexadecimal/binário, em cada um dos grupos que fez a afirmação (i.e., a 1ª parte da afirmação deles está correcta: *"têm o mesmo valor"*); mas (ii) ele contesta a afirmação de um dos grupos, e só poderá estar a referir-se à 2ª parte da afirmação: *"e portanto representam o mesmo valor decimal!"*

A questão resume-se a: em que situações é que a mesma codificação (em binário/hexadecimal) de 2 números pode não representar o mesmo valor no sistema de numeração decimal?

Pense um pouco antes de continuar!

Resposta: se esses números representarem tipos de dados distintos, e algo mais...
Neste caso em particular, sabe-se que um dos valores é um inteiro sem sinal (mais concretamente, um endereço de memória, que é o valor do apontador para a *stack*); se o outro registo *"foi usado para representar uma das variáveis inteiras do programa"*, é muito provável que essa variável tenha sido declarada como `int`.
Então, a 2ª questão que se pode colocar é a seguinte: em que situação é que a representação binária de um inteiro sem sinal é diferente da de um inteiro em complemento para 2? Apenas na representação de nºs negativos em complemento para 2! Se o nº for positivo e puder ser representado na notação em complemento para 2, então essa representação é igual à conversão directa de um nº decimal (sem sinal) num nº binário/hexadecimal.
Completando a resposta começada em cima: se esses números representarem tipos de dados distintos, e se forem ambos positivos.

b) Suponha que o 1º grupo via nos registos que tinham o mesmo conteúdo o valor `0x800c14`*, enquanto o segundo grupo tinha* `0xfffffffe`*. Calcule e indique o valor, em decimal, armazenado em cada um dos registos, nos 2 grupos.*

Sugestão de resolução

1º grupo

Passando *0x800c14* para binário com 32 bits: 0000 0000 1000 0000 0000 1100 0001 0100_2 ; representa um valor positivo (a variável inteira em `%eax`) ou um valor sem sinal (o apontador para o topo da *stack*), e ambos valem o mesmo:
$2^{23} + 2^{11} + 2^{10} + 2^4 + 2^2$

2º grupo

Passando *0xfffffffe* para binário com 32 bits: 1111 1111 1111 1111 1111 1111 1111 1110_2 ; representa um valor negativo pequeno (o que estará em `%eax`), ou um valor muito grande se for considerado sem sinal (o apontador para o topo da *stack*), e valem:
- em `%eax` : calculando o complemento para 2, este conjunto de bits vale (-10_2) ou (-2) em decimal;
- em `%esp`: o seu conteúdo é um endereço de memória que vale $(2^{32} + 2^{31} + ... + 2^3 + 2^2 + 2^1)$ ou, mais simplesmente, $(2^{32}-2)$.

B1

Imagine que está a desenvolver software (em C) para controlo de um equipamento industrial (que requer bastante precisão numérica nas coordenadas). Esse software, depois de devidamente testado num PC, vai ser portado para um outro processador (um microcontrolador) que ficará embebido no equipamento. Contudo, esse processador não suporta o formato de precisão dupla conforme especificado na norma IEEE 754, pelo que todas as variáveis que tenham sido declaradas como `double` *serão representadas apenas com 32 bits, no formato de precisão simples dessa mesma norma.*
Considere os seguintes valores, resultados intermediários de operações com variáveis que declarou como sendo do tipo `double`*.*
Mostre como serão representadas no processador/memória do sistema embebido. Comente os resultados obtidos.

Nota: os valores apresentados nas alíneas seguintes estão em hexadecimal, e foram obtidos com ferramentas de debugging, enquanto o software era testado num Pentium.

a) 0x 48 A9 01 00 80 C5 00 00

Sugestão de resolução

Há uma decisão de fundo a tomar antes de atacar este pb:
(i) vamos 1º identificar os limites de representação de nºs normalizados e desnormalizados (intervalo de valores e resolução) para cada um dos formatos (precisão simples/dupla) e depois analisar cada um dos valores propostos, ou
(ii) começamos de imediato a tentar calcular o valor decimal de cada um destes dados e tentamos convertê-los depois para o novo formato.

A 1ª abordagem dá-nos uma melhor perspectiva do que está em causa, mas exige mais trabalho no início; contudo, como esta questão tem 3 conjuntos de valores, este acréscimo de trabalho inicial é depois "amortizado" em cada uma das alíneas consequentes. Vamos pois seguir esta abordagem.

Um outro aspecto a analisar é tentar identificar, antes de qq outra actividade, que tipo de pb's poderão surgir quando tentamos converter um valor de um formato em FP para outro que utiliza menos bits, pois coloca-nos em melhor posição para uma análise crítica dos resultados que formos obtendo ao longo da resolução. Vamos ver o quero dizer com isto.

O que distingue estes 2 formatos é essencialmente o nº de bits usado para representar o expoente (8 e 11) e a mantissa (23 e 52).
Implicações:
- expoente: quanto maior o nº de bits usado no expoente maior é a gama de valores que é possível representar, quer no eixo positivo, quer no negativo; isto terá como consequência que (i) valores muito grandes (em modo absoluto) em precisão dupla poderão não ter representação em precisão simples (*overflow* positivo ou negativo, que de acordo com a norma IEEE será codificado como sendo $\pm \infty$), ou (ii) valores normalizados relativamente pequenos (em modo absoluto) em precisão dupla poderão ter de ser representados como desnormalizados em precisão simples, ou ainda que (iii) valores normalizados muito pequenos (em modo absoluto) em precisão dupla poderão não ter representação em precisão simples (*underflow* positivo ou negativo, que de acordo com a norma IEEE será codificado como sendo ± 0); se esta última situação ocorrer, então certamente todos os valores desnormalizados em precisão dupla serão considerados *underflow* (positivo ou negativo) e representados como ± 0, de acordo com a norma IEEE;
- mantissa: quanto maior o nº de bits usado na mantissa maior é a resolução do valor representado i.e., a sua representação possui um maior nº de dígitos significativos; na conversão de um valor noutro com menos bits para representar a mantissa, os dígitos em excesso são removidos (por arredondamento ou truncatura) e o resultado da conversão pode

ser um valor diferente do original, nomeadamente se forem removidos dígitos significativos diferentes de zero; isto é ainda mais crítico se o resultado da conversão for um valor desnormalizado, uma vez que este tem um nº variável de dígitos significativos (diminui consoante o valor se aproxima de zero). Questão para um candidato a **E**xcelente: quantos dígitos decimais significativos estão num valor em FP (formato normalizado) em precisão simples e em dupla?

Vamos então identificar os limites de representação de nºs normalizados e desnormalizados:
- precisão simples, caminhando de $-\infty$ até $+\infty$ (em caso de dúvida, consultar a resolução da alínea d) de A3)

]-2^128, -2^-126]]-2^-126, -2^-149] [2^-149, 2^-126[[2^-126, 2^128[

- - precisão dupla, caminhando de $-\infty$ até $+\infty$ (à semelhança da precisão simples, o expoente usa a notação excesso 2^{n-1}-1, i.e., 1023)

]-2^1024, -2^-1022]]-2^-1022, -2^-1074] [2^-1074, 2^-1022[[2^-1022, 2^1024[

Olhando para estes intervalos, já são mais claras a afirmações feitas anteriormente sobre as implicações do nº diferente de bits para representar o expoente; apenas alguns exemplos:
- qq nº em precisão dupla normalizado $\geq 2^{128}$ e $<2^{1024}$ é *overflow* positivo quando convertido para precisão simples;
- qq nº em precisão dupla normalizado $\geq 2^{-149}$ e $<2^{-126}$ é representado como desnormalizado quando convertido para precisão simples;
- qq nº em precisão dupla normalizado $\geq 2^{-1074}$ e $<2^{-149}$ é *underflow* positivo quando convertido para precisão simples;
- qq nº em precisão dupla desnormalizado é *underflow* (positivo ou negativo) quando convertido para precisão simples.

E agora vamos ver o caso concreto desta alínea (os bits do expoente estão sublinhados):
0x 48 A9 01 00 80 C5 00 00 = 0100 1000 1010 100 1000 1010___ 1001 0000 0001 0000 0000 1000 0000 1100 0101 0000 0000 0000 0000_2

S = 0 (positivo)
E = 100 1000 1010_2 (=> normalizado) = 1024+128+8+2 = (1024+138)
F = 1001 0000 0001 0000 0000 1000 0000 1100 0101 0000 0000 0000 0000_2

Donde,
$V = (-1)^S * 1.F * 2^{E-1023} = = (-1)^0 * 1.F * 2^{(1024+138)-1023} = 1.F * 2^{139}$ =>
=> não é possível representá-lo em precisão simples por estar fora da gama de valores representáveis (*overflow* positivo); vai ter de ser codificado como sendo $+\infty$

Resultado:
0111 1111 1000 0000 0000 0000 0000 0000_2 = **0x 7F 80 00 00**

Imagine que está a desenvolver software (em C) para controlo de um equipamento industrial (que requer bastente precisão numérica nas coordenadas). Esse software, depois de devidamente testado num PC, vai ser portado para um outro processador (um microcontrolador) que ficará embebido no equipamento. Contudo, esse processador não suporta o formato de precisão dupla conforme especificado na norma IEEE 754, pelo que todas as variáveis que tenham sido declaradas como `double` *serão representadas apenas com 32 bits, no formato de precisão simples dessa mesma norma.*
Considere os seguintes valores, resultados intermediários de operações com variáveis que declarou como sendo do tipo `double`*.*
Mostre como serão representadas no processador do sistema embebido. Comente os resultados obtidos.

Nota: os valores apresentados nas alíneas seguintes estão em hexadecimal, e foram obtidos com ferramentas de debugging, enquanto o software era testado num Pentium.

a) (...)
b) 0x 34 09 01 00 80 00 00 00

Sugestão de resolução

Após todos os considerandos da alínea anterior
0x 34 09 01 00 80 00 00 00 = 0011 0100 0000 011 0100 0000 1001 0000 0001 0000 0000 1000 0000 0000 0000 0000 0000 0000 0000_2

S = 0 (positivo)
E = 011 0100 0000_2 (=> normalizado) = 512+256+64 = 832
F = 1001 0000 0001 0000 0000 1000 0000 0000 0000 0000 0000 0000 0000_2

Donde,
$V = (-1)^S * 1.F * 2^{E-1023} = = (-1)^0 * 1.F * 2^{832-1023} = 1.F * 2^{-191}$ =>
=> não é possível representá-lo em precisão simples por estar fora da gama de valores representáveis (*underflow* positivo); vai ter de ser codificado como sendo +0

Resultado:
$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_2$ = **0x0**

Imagine que está a desenvolver software (em C) para controlo de um equipamento industrial (que requer bastente precisão numérica nas coordenadas). Esse software, depois de devidamente testado num PC, vai ser portado para um outro processador (um microcontrolador) que ficará embebido no equipamento. Contudo, esse processador não suporta o formato de precisão dupla conforme especificado na norma IEEE 754, pelo que todas as variáveis que tenham sido declaradas como `double` *serão representadas apenas com 32 bits, no formato de precisão simples dessa mesma norma.*
Considere os seguintes valores, resultados intermediários de operações com variáveis que declarou como sendo do tipo `double`*.*
Mostre como serão representadas no processador do sistema embebido. Comente os resultados obtidos.

Nota: os valores apresentados nas alíneas seguintes estão em hexadecimal, e foram obtidos com ferramentas de debugging, enquanto o software era testado num Pentium.

a) (...)
b) (...)
c) 0x B7 41 90 10 80 00 C0 00

Sugestão de resolução

Após todos os considerandos da alínea a)
0x B7 41 90 10 80 00 C0 00 = 1011 0111 0100 011 0111 0100 0001 1001 0000 0001 0000 1000 0000 0000 0000 0000 1100 0000 0000_2

S = 1 (negativo)
E = 011 0111 0100_2 (=> normalizado) = 512+256+64+32+16+4 = 884
F = 0001 1001 0000 0001 0000 1000 0000 0000 0000 0000 1100 0000 0000_2

Donde,
$V = (-1)^S * 1.F * 2^{E-1023} = = (-1)^1 * 1.F * 2^{884-1023} = 1.0001\ 1001\ 0000\ 0001\ 0000\ 1000\ 0000\ 0000\ 0000\ 0000\ 11_2 * 2^{-139}$ =>
=> representável em precisão simples, mas como valor desnormalizado.

Então, vai ser preciso modificar a expressão acima apresentada, para que ela fique semelhante à expressão que nos dá o valor de um nº desnormalizado em precisão simples:
$V = (-1)^S * 0.F * 2^{-126} = -1.0001\ 1001\ 0000\ 0001\ 0000\ 1000\ 0000\ 0000\ 0000\ 0000\ 11_2 * 2^{-139}$
=
$= -0.0000\ 0000\ 0000\ 1000\ 1100\ 1000\ 0010\ 0001\ 0000\ 0000\ 0000\ 0001\ 1_2 * 2^{13} * 2^{-139}$ =
(como só existem 23 bits na mantissa em precisão simples, vão-se perder dígitos significativos $\neq 0$ na conversão...)
$= -0.0000\ 0000\ 0000\ 1000\ 1100\ 100_2 * 2^{-126}$

Resultado:
$1000\ 0000\ 0000\ 0000\ 0000\ 0100\ 0110\ 0100_2$ = **0x 80 00 04 64** (com perda de dígitos significativos)

Organização e funcionamento dum computador

A1

Considere a execução da seguinte instrução (16-bits) num processador de 16-bits (little-endian) e usando barramentos de 16 bits:

- soma de 2 inteiros em registos (R3 e R4), sendo o resultado guardado no topo da stack, sem destruir o que lá estava.

Antes da execução, considere os seguintes conteúdos dos registos deste processador e de parte da sua memória:

- em registos: IP (0x8080), SP (0x8414), R3 (0x8408), R4 (0x88);

- na memória: de 0x8080 a 0x8083 (0C 62 14 FF); de 0x8410 a 0x8417 (09 10 2A 3B 4C 5D 6E 7F)

a) Identifique o conjunto de sinais que deverão estar em cada um dos barramentos que ligam o processador à memória, indicando a dimensão de cada barramento e a função de cada um desses conjuntos de sinais.

Sugestão de resolução

O processador comunica com a memória com 2 objectivos principais: (i) para ir buscar instruções (há quem defenda uma memória só para instruções separada de uma outra só para dados) e (ii) para ir buscar ou para armazenar dados, na consequência da execução da própria instrução, depois de trazida da memória e descodificada.
Quer num caso, quer no outro, o processador necessita de especificar (i) a localização da memória que pretende aceder e (ii) o tipo de acesso: se é para ler o que está na memória ou se é para escrever na memória; para além disso, (iii) a informação propriamente dita tem de ser transportada do processador para a memória, ou da memória para o processador.

Resumindo:

- (i) para especificar a localização da memória que se pretende aceder, usa-se o barramento de endereços que, neste caso, é de 16 bits, conforme dito no enunciado; é assim possível especificar um endereço de memória na gama [0, 2^16-1];
- (ii) o tipo de acesso será de leitura (sinal RD) ou de escrita (sinal WR); são estes os únicos 2 sinais do barramento de controlo claramente especificados na documentação sobre esta matéria;
- (iii) a informação - que tanto pode ser a instrução a executar, como os dados a ler/escrever da/na memória - são transportados pelo barramento de dados que, neste caso, é de 16 bits, conforme dito no enunciado.

Considere a execução da seguinte instrução (16-bits) num processador de 16-bits (little-endian) e usando barramentos de 16 bits:

- soma de 2 inteiros em registos (R3 e R4), sendo o resultado guardado no topo da stack, sem destruir o que lá estava.

Antes da execução, considere os seguintes conteúdos dos registos deste processador e de parte da sua memória:

- em registos: IP (0x8080), SP (0x8414), R3 (0x8408), R4 (0x88);

- na memória: de 0x8080 a 0x8083 (0C 62 14 FF); de 0x8410 a 0x8417 (09 10 2A 3B 4C 5D 6E 7F)

a) (...)

b) Apresente cronologicamente em hexadecimal (e explicitando o raciocínio seguido) o conjunto de valores que deverá circular nos barramentos durante a execução desta instrução (cujo início ocorreu quando o processador terminou a execução da instrução anterior).

Sugestão de resolução

Cronologicamente:
- para ir buscar a instrução: (i) o processador coloca o conteúdo do IP no barramento de endereços (0x8080) e activa o sinal de RD; (ii) a memória coloca o conteúdo das células em 0x8080 e 0x8081 no barramento de dados (0x0C62);
- para ir guardar o resultado da operação de soma (os operandos fonte já estavam no processador, nos registos R3 e R4): (i) o processador actualiza o conteúdo do SP (põe-no a apontar para a posição da memória logo acima do topo da *stack*, deixando espaço para colocar lá um valor de 16 bits), coloca esse valor no barramento de endereços (0x8412) e activa o sinal de WR; (ii) o processador coloca o resultado da soma (0x8408+0x88) no barramento de dados (0x8490).

Considere a execução da seguinte instrução (16-bits) num processador de 16-bits (little-endian) e usando barramentos de 16 bits:
- soma de 2 inteiros em registos (R3 e R4), sendo o resultado guardado no topo da stack, sem destruir o que lá estava.
Antes da execução, considere os seguintes conteúdos dos registos deste processador e de parte da sua memória:
- em registos: IP (0x8080), SP (0x8414), R3 (0x8408), R4 (0x88);
- na memória: de 0x8080 a 0x8083 (0C 62 14 FF); de 0x8410 a 0x8417 (09 10 2A 3B 4C 5D 6E 7F)

a) (...)
b) (...)
c) Indique, justificando, o conteúdo dos 4 registos referidos no enunciado (em hexadecimal), após a execução desta instrução.

Sugestão de resolução

Registos que foram modificados com a execução da instrução: IP (passa a apontar para a nova instrução, que se encontra a 2 células de distância) e SP que foi actualizado para apontar para o novo valor no topo da *stack*); os restantes 2 não foram modificados, pois foram apenas apenas usados para leitura.

Assim, após a execução:
- IP - 0x8082
- SP - 0x8412
- R3 e R4 - mantêm o que tinham antes.

Considere a execução da seguinte instrução (16-bits) num processador de 16-bits (little-endian) e usando barramentos de 16 bits:
- soma de 2 inteiros em registos (R3 e R4), sendo o resultado guardado no topo da stack, sem destruir o que lá estava.
Antes da execução, considere os seguintes conteúdos dos registos deste processador e de parte da sua memória:
- em registos: IP (0x8080), SP (0x8414), R3 (0x8408), R4 (0x88);
- na memória: de 0x8080 a 0x8083 (0C 62 14 FF); de 0x8410 a 0x8417 (09 10 2A 3B 4C 5D 6E 7F)

a) (...)
b) (...)
c) (...)
d) Indique, justificando, o endereço e conteúdo das células de memória que foram modificadas com a execução desta instrução (em hexadecimal).

Sugestão de resolução

Apenas 2 células de memória foram modificadas: aquelas que foram escritas com o valor de 16 bits do resultado da soma de R3 com R4 (e que já vimos foi 0x8490); essas células encontram-se em 0x8412 e 0x8413.

Assim:
- de 0x8412 a 0x8413 : 90 84

R1.
Comente as seguintes afirmações, referindo claramente as incorrecções que encontrar e corrigindo-as:

a) "O barramento de controlo num computador serve para transportar as instruções da memória para o processador, colocando-as num registo apontado pelo IP, para o processador as executar".

Sugestão de resolução

O barramento de controlo contém um conjunto de sinais de apoio ao controlo das operações de transferência de informação de uns lados para outros no interior de um computador. Nas relações entre o processador e a memória, os principais sinais são os que especificam o tipo de operação a ser efectuado na memória, quando esta recebe um valor no barramento de endereços: se é uma operação de leitura da memória, ou se é uma operação de escrita na memória.
Neste caso concreto, de busca de uma instrução, o barramento de controlo é usado para indicar à memória que o processador pretende ler o conteúdo de uma (ou mais) células de memória.

Adicionalmente, as instruções quando chegam ao processador vindas da memória, para serem executadas, são colocadas num registo (é verdade, normalmente designado pelo registo de instrução, ou IR), mas o IP não aponta normalmente para esse registo, mas sim para a posição de memória que tem a próxima instrução a ser executada.

Comente as seguintes afirmações, referindo claramente as incorrecções que encontrar e corrigindo-as:

a) (...)
b) "A análise da estrutura interna de um processador mostra que ele possui um bloco de hardware denominado de 'instruction set', que serve para converter para binário o programa com os comandos para o CPU, codificado em ASCII".

Sugestão de resolução

O *'instruction set*' é uma especificação do conjunto de instruções que a unidade de controlo de um processador é capaz de descodificar e interpretar, e não um bloco de *hardware*!
As instruções que o processador vai buscar à memória para executar, "*os comandos para o CPU*", estão já codificadas de acordo com essa especificação, i.e., o conjunto de bits que constitui uma instrução em "linguagem máquina" está já organizado de acordo com o formato de instrução que foi explicitamente especificado para cada uma das instruções do *instruction set* de um processador.
Estas instruções não podem estar codificadas em ASCII, pois a sua execução pelo processador seria bastante ineficiente. Compete a um programa montador (*assembler*) pegar no ficheiro de texto (programa em *assembly*) que contém mnemónicas que descrevem textualmente as instruções que o processador vai executar, e converter esse ficheiro de texto (que está codificado em binário de acordo com a tabela ASCII) para um ficheiro em binário de acordo com o o formato de instruções específico desse *instruction set*.

Comente as seguintes afirmações, referindo claramente as incorrecções que encontrar e corrigindo-as:

a) (...)
b) (...)

c) "Ao compilar o ficheiro `hello.c`, vou obter o ficheiro `hello.s`, que é um programa em assembly codificado em binário, de acordo com a definição dos formatos de instrução do 'instruction set' do processador onde está a ser executado o compilador".

Sugestão de resolução

O ficheiro `hello.s` é efectivamente "*um programa em assembly codificado em binário*", mas... o que está codificado em binário são os caracteres desse ficheiro (de texto!), e de acordo com a tabela ASCII.

Só depois de passar por um *assembler* é que esse programa estará num formato "*em binário, de acordo com a definição dos formatos de instrução do 'instruction set' do processador*" onde irá ser executado. Teremos então um ficheiro objecto; mais concretamente, um *relocatable object program*, conforme descrito na secção 3.1 do texto de apoio ISC disponibilizado.

E este programa até poderá depois ser executado num processador diferente daquele onde está a ser executado o compilador.
Num caso como este estaremos na presença de um compilador cruzado (*cross-compiler*), e esta situação é muito comum quando se pretende desenvolver *software* para microcontroladores (por ex.) usando a plataforma do PC como sistema de desenvolvimento.

Comente as seguintes afirmações, referindo claramente as incorrecções que encontrar e corrigindo-as:

a) (...)
b) (...)
c) (...)
d) "Durante a execução de um programa, compete ao sistema operativo transferir cada uma das instruções da memória para o CPU, para serem executadas".

Sugestão de resolução

O sistema operativo é constituído por um conjunto de programas que são executados no processador da mesma maneira que os programas desenvolvidos por um qualquer utilizador. Poderão conter instruções especiais extra e alguns outros privilégios (particularmente nas permissões de acesso a recursos), mas fora isto são programas como os outros.

A unidade de controlo do processador é que é o principal responsável por controlar as operações que transferem as "*instruções da memória para o CPU*".
Assim, compete a este bloco de *hardware* gerar os sinais que, numa sequência cronológica apropriada, (i) colocarão o conteúdo do IP no barramento de endereços, (ii) activarão o sinal de controlo de leitura da memória (RD) e que depois (iii) colocarão o que estiver no barramento de dados no registo de instrução (IR) do processador; posteriormente, a unidade de controlo irá interpretar os bits que estiverem no IR (i.e., descodificar a instrução).

Por curiosidade: nem todos os processadores têm um registo de instrução com dimensão fixa; quando os processadores têm instruções com comprimento variável, os processadores têm um "registo" organizado como uma fila de espera com possibilidade de armazenar vários bytes. Por exemplo, uma das diferenças entre o processador i8086 e o i8088 (do PC original) era a dimensão desta fila de espera: 6 *bytes* no i8088 e 8 *bytes* no i8086.

ISA do IA-32

A1

No interior de uma função em C encontra-se a seguinte instrução:

```
var_x += elem_y[ind]
```

Quando essa instrução é compilada num PC (little endian) com "`gcc -S`" as únicas linhas de código que ele gera relacionadas com essa instrução em C, são:

```
movl    8(%ebp), %ecx
addl    -24(%ebp, %ecx,4), %edi
```

O array `elem_y` *é a única variável local desta função, e que ocupa um espaço de memória contíguo à célula cujo endereço é o actual valor de* `%ebp`*; por outro lado, o índice do array é a única variável cujo valor foi passado para esta função como argumento.*

O estado do computador imediatamente antes da execução destas 2 instruções é dado pelos seguintes valores presentes em registos e na memória (valores obtidos com vários comandos `x` *do depurador* `dbg`*):*

Reg	Valor	Reg	Valor
%eax	0x855	%esi	0x12
%ebx	0	%edi	0x810
%ecx	0xff	%ebp	0x8401020
%edx	1	%esp	0x8401008
		%eip	0x812401e

Endereço / ?	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
0x812401?	45	04	89	ec	5d	c3	8d	76	00	00	00	55	89	e5	8b	4d
0x840100?	00	2f	ff	ff	12	20	40	08	02	04	00	00	fc	ff	ff	ff
0x840101?	f4	ff	ff	ff	00	80	ff	ff	02	02	00	00	00	00	08	00
0x840102?	30	10	40	08	02	00	fc	ff	02	00	00	00	12	0e	bc	00

a) Relativamente ao array `elem_y` *indique, justificando: (i) a sua dimensão (quantos elementos), (ii) a dimensão de cada um dos elementos do array (quantos bytes), (iii) o valor do 1º elemento do array (em decimal), e (iv) o valor do índice (em decimal), imediatamente antes de executar a instrução em C.*

Sugestão de resolução

Analisando a instrução em C, o que é que ela indica?

Uma adição do elemento do *array* `elem_y` cujo índice é a variável `ind`, com a variável `var_x`.

Ainda de acordo com o enunciado, sabe-se que o *array* elem_y se encontra na memória: um bloco de células contíguas desde uma localização inicial até à célula junta àquela que está a ser apontada pelo conteúdo do registo %ebp. E sabe-se tb onde se encontra armazenado (na memória) o valor da varíavel que representa o índice do *array*.
Contudo, o enunciado não faz qq referência explícita à localização da variável `var_x`; mas será preciso? Qual deverá ser o local mais apropriado para conter o seu valor? E o que é que nos indica o código em *assembly*? Julgo não ser preciso dizer mais nada...
O código em *assembly* diz-nos tudo o resto que precisamos de saber...

Com base nesta informação, se ainda não conseguiu resolver, tente de novo...

O código em *assembly* diz-nos tudo o que precisamos de saber: que cada elemento do *array* ocupa 4 células na memória, logo a **dimensão de cada elemento é de 4 *bytes*** (basta ver o factor de escala no modo de endereçamento à memória na operação de adição), e que portanto a **dimensão do *array* é de** (24 *bytes* / 4 *bytes* =) **6 elementos**.
Sabendo onde começa o *array* (em `Mem[(%ebp)-24]` = `Mem[0x8401020-24]` = `Mem[0x8401020-0x18]` = `Mem[0x8401008]`), é possível retirar da tabela os 4 *bytes* que constituem o valor do 1º elemento (em hex `02 04 00 00` `02 04 00 00` , e lendo esse nº como *little endian*, `0x402`) e calcular o seu valor em decimal (em complemento para 2 é positivo, portanto basta converter de base 16 ou 2 para base 10; logo, dá 1K+2, **1026**).
Para saber o valor do índice antes da execução, basta analisar o conteúdo da memória nos endereços onde ele se encontra (de `0x8401020+8` a `0x8401020+11`), e passá-lo a decimal (sem esquecer a questão de *little endian*): **2**.

No interior de uma função em C encontra-se a seguinte instrução:

```
var_x += elem_y[ind]
```

Quando essa instrução é compilada num PC (little endian) com "`gcc -S`" ... :

```
movl   8(%ebp), %ecx
addl   -24(%ebp, %ecx,4), %edi
```

Reg	Valor	Reg	Valor
%eax	0x855	%esi	0x12
%ebx	0	%edi	0x810
%ecx	0xff	%ebp	0x8401020
%edx	1	%esp	0x8401008
		%eip	0x812401e

Endereço / ?	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
0x812401?	45	04	89	ec	5d	c3	8d	76	00	00	00	55	89	e5	8b	4d
0x840100?	00	2f	ff	ff	12	20	40	08	02	04	00	00	fc	ff	ff	ff
0x840101?	f4	ff	ff	ff	00	80	ff	ff	02	02	00	00	00	00	08	00
0x840102?	30	10	40	08	02	00	fc	ff	02	00	00	00	12	0e	bc	00

a) (...)
b) Indique o valor da variável `var_x` após a execução destas 2 instruções em assembly. Explicite o raciocínio/cálculo efectuado.
*Nota: a resolução correcta desta alínea apenas é exigida para a classificação **R**.*

Sugestão de resolução

Este pb limita-se a pedir a análise da execução de 2 simples instruções: uma de transferência de info (Mem->Reg), outra uma adição em que um dos operandos está em memória.

A 1ª vai buscar o valor da variável de indexação do *array* (`ind`) e coloca-o no registo `%ecx`. (que já se tinha visto na alínea anterior que era o valor 2, i.e., o índice do 3º elemento do *array*).

A 2ª instrução vai somar esse 3º elemento do *array* (em memória, nas 4 células que começam em `Mem[(%ebp)-24+4*(%ecx)]` = `Mem[0x8401020-24+4*2]` = `Mem[0x8401020-16]` = `Mem[0x8401020-0x10]` = `Mem[0x8401010]`), que é o valor `0xfffffff4`, com o valor que estava na `var_x` (no registo `%edi`).

É óbvio que quando se pede o valor de uma variável dum programa em C (do tipo `int`, como muito provavelmente é o caso), se espera que o resultado seja apresentado em decimal... Neste caso, feitas as contas (de cabeça), o resultado deverá ser... **2052**.)

Não se esqueça de indicar sempre as operações que efectuar, para garantir que: (i) se se enganou apenas na aritmética, poder receber a cotação quase integral da resolução do exercício, e (ii) o resultado foi obtido por si, e não estava a "circular" pela sala.

No interior de uma função em C encontra-se a seguinte instrução:

```
var_x += elem_y[ind]
```

Quando essa instrução é compilada num PC (little endian) com "`gcc -S`" ... :

```
movl    8(%ebp), %ecx
addl    -24(%ebp, %ecx,4), %edi
```

Reg	Valor	Reg	Valor
%eax	0x855	%esi	0x12
%ebx	0	%edi	0x810
%ecx	0xff	%ebp	0x8401020
%edx	1	%esp	0x8401008
		%eip	0x812401e

Endereço / ?	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
0x812401?	45	04	89	ec	5d	c3	8d	76	00	00	00	55	89	e5	8b	4d
0x840100?	00	2f	ff	ff	12	20	40	08	02	04	00	00	fc	ff	ff	ff
0x840101?	f4	ff	ff	ff	00	80	ff	ff	02	02	00	00	00	00	08	00
0x840102?	30	10	40	08	02	00	fc	ff	02	00	00	00	12	0e	bc	00

a) (...)
b) (...)
c) Suponha que a variável `var_x` se encontra em memória, cujo endereço é o conteúdo de `%edi` . A instrução de adição (em assembly) poderia ser substituída por `addl -24(%ebp,%ecx,4),(%edi)` ? Justifique.
*Nota: a resolução correcta desta alínea apenas é exigida para a classificação **R**.*

Sugestão de resolução

A instrução proposta para substituir, como funciona? ...

A arquitectura dos processadores compatíveis com a linha i8086 suporta este tipo de instruções aritméticas, em que os 2 operandos que são explicitados numa única instrução, estão localizados em memória?

No interior de uma função em C encontra-se a seguinte instrução:

```
var_x += elem_y[ind]
```

Quando essa instrução é compilada num PC (little endian) com "`gcc -S`" ... :

```
movl    8(%ebp), %ecx
addl    -24(%ebp, %ecx,4), %edi
```

Reg	Valor	Reg	Valor
%eax	0x855	%esi	0x12
%ebx	0	%edi	0x810
%ecx	0xff	%ebp	0x8401020
%edx	1	%esp	0x8401008
		%eip	0x812401e

Endereço / ?	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
0x812401?	45	04	89	ec	5d	c3	8d	76	00	00	00	55	89	e5	8b	4d
0x840100?	00	2f	ff	ff	12	20	40	08	02	04	00	00	fc	ff	ff	ff
0x840101?	f4	ff	ff	ff	00	80	ff	ff	02	02	00	00	00	00	08	00
0x840102?	30	10	40	08	02	00	fc	ff	02	00	00	00	12	0e	bc	00

a) (...)
b) (...)
c) (...)
d) Se o processador do PC tivesse a arquitectura típica dum processador RISC, o compilador não geraria a 2ª linha de código apresentada no enunciado. Mostre, justificando, que código teria sido gerado (use a sintaxe do assembler do IA-32).
*Nota: a resolução correcta desta alínea apenas é exigida para a classificação **B**.*

Sugestão de resolução

Uma das regras básicas de qq arquitectura RISC é que cada instrução apenas efectua uma operação elementar: ou de acesso à memória - para ler ou escrever - ou uma operação lógica/aritmética.
Por isso, estas arquitecturas são tb por vezes conhecidas por arquitecturas de *load*/*store*, pois são estas as únicas instruções de acesso à memória. Assim, numa adição, os operandos fonte têm de estar já em registos, e o resultado terá de ficar armazenado tb em registo.
E é possível especificar os 3 operandos (2 fonte, e um destino) em qq operação lógica/aritmética; usando uma sintaxe semelhante à do *assembler* da Gnu, uma soma seria escrita como `addl  Src1, Src2, Dest` .
Adicionalmente, as arquitecturas RISC suportam apenas um leque muito limitado de modos de especificação de um endereço de memória: ou apenas especificam uma localização de memória através da indicação de um registo e de um deslocamento (*offset*) em relação a esse registo, ou então suportam adicionalmente a possibilidade de especificar um endereço como sendo a soma do conteúdo de 2 registos.

Com base nestes dados, é possível chegar a uma versão desse código (**tente primeiro antes de espreitar...**):

```
multl   %ecx, 4, %edx
addl    %edx, %ebp, %edx
movl    -24(%edx), %edx
addl    %edx, %edi, %edi
```

No interior de uma função em C encontra-se a seguinte instrução:

```
var_x += elem_y[ind]
```

Quando essa instrução é compilada num PC (little endian) com "`gcc -S`" ... :

```
movl    8(%ebp), %ecx
addl    -24(%ebp, %ecx,4), %edi
```

Reg	Valor	Reg	Valor
%eax	0x855	%esi	0x12
%ebx	0	%edi	0x810
%ecx	0xff	%ebp	0x8401020
%edx	1	%esp	0x8401008
		%eip	0x812401e

Endereço / ?	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
0x812401?	45	04	89	ec	5d	c3	8d	76	00	00	00	55	89	e5	8b	4d
0x840100?	00	2f	ff	ff	12	20	40	08	02	04	00	00	fc	ff	ff	ff
0x840101?	f4	ff	ff	ff	00	80	ff	ff	02	02	00	00	00	00	08	00
0x840102?	30	10	40	08	02	00	fc	ff	02	00	00	00	12	0e	bc	00

a) (...)
b) (...)
c) (...)
d) (...)
e) Indique, justificando, qual deverá ser o conteúdo da célula de memória em `0x8124020`*.*
Nota: a resolução correcta desta alínea apenas é exigida para a classificação ***B****.*

Sugestão de resolução

Qual a relação deste endereço com o resto do enunciado do pb?
Não adianta avançar enquanto não responder a esta questão...

Este endereço é um valor muito próximo do valor do `%eip` imediatamente antes da execução destas 2 instruções; mais concretamente, está a apontar para o 3º *byte* localizado após o 1º *byte* da 1ª instrução.
Portanto, ou aponta para uma célula de memória que ainda contém parte da 1ª instrução, ou provavelmente é um endereço de memória que contém parte da 2ª instrução.

Sem ter de consultar o manual de referência da Intel com a especificação de todas as instruções do *instruction set* da família IA-32, será possível chegar a alguma conclusão?
Tente...!

Analisemos essa 1ª instrução: é uma instrução de `mov`, de Mem->Reg. De acordo com os formatos de instrução da Intel, qq instrução deste tipo em que especificam 2 operandos, são necessários pelo menos 2 *bytes*: um contendo o código de operação (*opcode*), e outro especificando um registo **e** um registo/memória.
Se o campo da instrução que especifica o registo/memória indicar uma localização de memória que necessite de adicionar o valor de um deslocamento (*offset*), este terá de vir no(s) *byte(s)* seguinte(s).

No caso concreto desta instrução, a especificação do endereço de memória - `8(%ebp)` - requer o valor de um deslocamento positivo e menor que 127, pelo que é possível especificá-lo usando apenas 1 *byte* adicional, que será o 3º *byte* da instrução. E este 3º *byte* encontra-se precisamente no endereço `0x8124020`...

Assim, sendo, qual será o valor que se encontra na memória nesse endereço?...

No interior de uma função em C encontra-se a seguinte instrução:

```
var_x += elem_y[ind]
```

Quando essa instrução é compilada num PC (little endian) com "`gcc -S`" ... :

```
movl   8(%ebp), %ecx
addl   -24(%ebp, %ecx,4), %edi
```

Reg	Valor	Reg	Valor
%eax	0x855	%esi	0x12
%ebx	0	%edi	0x810
%ecx	0xff	%ebp	0x8401020
%edx	1	%esp	0x8401008
		%eip	0x812401e

Endereço / ?	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
0x812401?	45	04	89	ec	5d	c3	8d	76	00	00	00	55	89	e5	8b	4d
0x840100?	00	2f	ff	ff	12	20	40	08	02	04	00	00	fc	ff	ff	ff
0x840101?	f4	ff	ff	ff	00	80	ff	ff	02	02	00	00	00	00	08	00
0x840102?	30	10	40	08	02	00	fc	ff	02	00	00	00	12	0e	bc	00

a) (...)
b) (...)
c) (...)
d) (...)
e) (...)
f) Face aos elementos disponibilizados no início deste enunciado, a seguinte afirmação estará correcta?
"O array `elem_y` é a única variável local desta função, e que ocupa um espaço de memória contíguo à célula cujo endereço é o actual valor de `%ebp`; por outro lado, o índice do array é a única variável cujo valor foi passado para esta função como argumento."
*Nota: a resolução correcta desta alínea, com consulta, está ao alcance da classificação **R**; sem consulta, apenas é exigida para a classificação **B**.*

Sugestão de resolução

Quando se analisa a estrutura da *stack* na fase de arranque de uma função, quais as tarefas que normalmente são efectuadas?

PENSE!

E na salvaguarda de registos, que registos competem à função chamada guardar?
E que registo é usado para escrita na 2ª instrução em *assembly* deste enunciado?
E onde vai ser salvaguardado o conteúdo desse registo?

E se ficar aí, o *array* pode ficar onde se indica no enunciado?...

a) (...)
b) (...)
c) (...)
d) (...)
e) (...)
f) (...)
g) Indique o endereço provável da instrução que invoca esta função.
*Nota: a resolução correcta desta alínea, após resolução da seguinte, pode estar ao alcance da classificação **R** (quase **B**); sem esta dica, é indispensável para a classificação **E**.*

Sugestão de resolução

Veja no fim da resolução da alínea seguinte.

No interior de uma função em C encontra-se a seguinte instrução:

```
var_x += elem_y[ind]
```

Quando essa instrução é compilada num PC (little endian) com "`gcc -S`" ... :

```
movl   8(%ebp), %ecx
addl   -24(%ebp, %ecx,4), %edi
```

Reg	Valor	Reg	Valor
%eax	0x855	%esi	0x12
%ebx	0	%edi	0x810
%ecx	0xff	%ebp	0x8401020
%edx	1	%esp	0x8401008
		%eip	0x812401e

Endereço / ?	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
0x812401?	45	04	89	ec	5d	c3	8d	76	00	00	00	55	89	e5	8b	4d
0x840100?	00	2f	ff	ff	12	20	40	08	02	04	00	00	fc	ff	ff	ff
0x840101?	f4	ff	ff	ff	00	80	ff	ff	02	02	00	00	00	00	08	00
0x840102?	30	10	40	08	02	00	fc	ff	02	00	00	00	12	0e	bc	00

a) (...)
b) (...)
c) (...)
d) (...)
e) (...)
f) (...)
g) (...)
h) Mostre a estrutura da stack associada a esta função na arquitectura IA-32, após a invocação da função e imediatamente antes da execução do corpo da função, indicando a dimensão e conteúdo de todas as células relevantes, bem como a localização do stack pointer e do base pointer.

Sugestão de resolução

De acordo com o enunciado, se o *array* é a única variável local da função e o seu índice é a única variável cujo valor foi passado como argumento, então pode-se assumir que `var_x` seja a única variável passada como referência, sendo provavelmente o segundo argumento.
Ainda de acordo com o enunciado, vamos tb assumir que esta função não vai salvaguardar o conteúdo do registo `%edi`.

Nestas condições, a *stack frame* associada a esta função tem a seguinte constituição e estrutura (pela ordem em que ela foi criada):
- de `Mem[(%ebp)+12]` a `Mem[(%ebp)+15]`, deverá estar o 2º argumento, com 4 *bytes*; valor: consultar a tabela acima e extrair `0xbc0e12`;
- de `Mem[(%ebp)+8]` a `Mem[(%ebp)+11]`, deverá estar o 1º argumento, com 4 *bytes*; valor: consultar a tabela acima e extrair `2`;
- de `Mem[(%ebp)+4]` a `Mem[(%ebp)+7]`, deverá estar o endereço de retorno, com 4 *bytes*; valor: consultar a tabela acima e extrair `0xfffc0002`;
- de `Mem[(%ebp)]` a `Mem[(%ebp)+3]`, deverá estar o valor do anterior *frame pointer*, com 4 *bytes*; valor: consultar a tabela acima e extrair `0x8401030`;
- de `Mem[(%ebp)-24]` a `Mem[(%ebp)-1]`, deverão estar os 6 inteiros do *array*, com 24 *bytes*; valores: consultar a tabela acima e extrair `...`;

Quanto aos valores dos registos `%ebp` e `%esp`: no 1º está o valor indicado na tabela do enunciado; o 2º deverá estar a apontar para o topo da *stack*, que nessa altura estará a apontar para o endereço reservado para o início do 1º elemento do *array*; ou seja, o `%esp` registo terá o valor `(%ebp)-24` (faça as contas...)

Quanto à alínea anterior: se o endereço de retorno é `0xfffc0002`, e se este endereço está a apontar para a instrução imediatamente após a instrução de invocação da função, resta saber qual a dimensão desta instrução; se apenas 2 *bytes*, ou se 5 *bytes* (consoante a distância a que se encontra a função, se é possível representar essa distância com apenas 8 bits, ou se serão precisos 32 bits).
Neste caso, 8 bits parecem ser insuficientes. Então o mais provável é a instrução estar localizada em `0xfffbfffd` (=`0xfffc0002-5`).

R1.
Considere a expressão aritmética com inteiros `a=b[i]-10`, inserida no corpo de uma função em C, em que `a` é uma variável global escalar (cujo apontador foi passado como 1º argumento), `b` é um array global (cujo apontador foi passado como 2º argumento), `i` é a única variável escalar da função.

a) Apresente o código assembly que poderia ser gerado pelo `gcc` para IA-32, quando encontrasse essa expressão.

Sugestão de resolução

A operação pretendida é uma subtracção, que no caso do IA-32 será do tipo `subl Src1,Dest`, o que corresponde a `Dest=Dest-Src` .
Isto sugere, desde já, que se carregue o elemento do *array* da memória para um registo, e se faça a subtracção com operandos em registos, para minimizar acessos à memória.
De notar ainda que o índice do *array*, sendo a única variável escalar da função, muito provavelmente estará alojada num registo (tb para minimizar acessos à memória); vamos considerar que é o registo `%ecx`.

Sugestão de conversão desta instrução em C para uma sequência de outras que possam ser directamente traduzidas para *assembly* do IA-32 (notar que a utilização de uma variável temporária, local à função, pressupõe que o compilador irá usar um registo para essa variável):

```
temp=b[i];
temp-=10;
a=temp;
```

Vamos ver agora se é possível traduzir cada uma destas instruções directamente para *assembly* (**tente antes de espreitar para a resolução...**):

Não é possível... pois é preciso ir tb à memória buscar os endereços do início do *array* e da variável global `a`.

```
        movl    12(%ebp), %esi          ; %esi= endereço do início
do array b
        movl    (%esi,%ecx,4), %eax     ; %eax= b[i]
        subl    $-10, %eax              ; %eax= b[i]-10
        movl    8(%ebp), %esi           ; %esi= endereço da variável
a
        movl    %eax, (%esi)            ; a= b[i]-10
```

Considere a expressão aritmética com inteiros `a=b[i]-10`, inserida no corpo de uma função em C, em que `a` é uma variável global escalar (cujo apontador foi passado como 1º argumento), `b` é um array global (cujo apontador foi passado como 2º argumento), `i` é a única variável escalar da função.

a) (...)
b) Mostre a estrutura da stack associada a esta função na arquitectura IA-32, após a invocação da função e imediatamente antes da execução do corpo da função, indicando a dimensão e conteúdo de todas as células relevantes, bem como a localização do stack pointer e do base pointer.

Sugestão de resolução

Esta questão é idêntica à do problema anterior. A única complexidade é saber se há ou não registos a salvaguardar nesta função.
Se se considerar que o `gcc` geraria o código apresentado na alínea anterior, então saberíamos que esta função necessita de usar o registo `%esi`, e que portanto este deverá ser salvaguardado na *stack*, entre o local onde se encontra o antigo *frame poiter*, e as variáveis locais (que não parece que existam nesta função, pelo menos a usarem a *stack*).

Assim, teríamos:
- de `Mem[(%ebp)+12]` a `Mem[(%ebp)+15]`, deverá estar o 2º argumento, o apontador para o *array* global com 4 *bytes*;
- de `Mem[(%ebp)+8]` a `Mem[(%ebp)+11]`, deverá estar o 1º argumento, a variável `a`, do tipo `int`, com 4 *bytes*;
- de `Mem[(%ebp)+4]` a `Mem[(%ebp)+7]`, deverá estar o endereço de retorno, com 4 *bytes*;
- de `Mem[(%ebp)]` a `Mem[(%ebp)+3]`, deverá estar o valor do anterior *frame pointer*, com 4 *bytes*;
- de `Mem[(%ebp)-4]` a `Mem[(%ebp)-1]`, deverá estar o conteúdo que o registo `%esi` tinha no início da execução da função, com 4 *bytes*.

Quanto aos valores dos registos `%ebp` e `%esp`: o 1º aponta para a célula que tem o *byte* menos significativo do antigo valor do *frame pointer*, enquanto que o 2º aponta para o *byte* menos significativo do conteúdo do registo `%esi`, que foi salvaguardado por esta função.

Considere a expressão aritmética com inteiros `a=b[i]-10`, inserida no corpo de uma função em C, em que `a` é uma variável global escalar (cujo apontador foi passado como 1º argumento), `b` é um array global (cujo apontador foi passado como 2º argumento), `i` é a única variável escalar da função.

a) (...)
b) (...)
c) Considere agora que o corpo desta função era apenas constituída por um ciclo `for` contendo no interior aquela expressão seguida da impressão do valor de `a`. A variável de controlo do ciclo é `i` (inicializada a 0) e o ciclo percorre todo o array (a dimensão do array, em termos de nº de elementos, é o 3º argumento passado para esta função).
Mostre o código assembly que o `gcc` iria gerar (com optimização `-O2`), colocando em comentários o código C correspondente (na versão `goto`).
Sugestão: escreva primeiro o código C da função.

Sugestão de resolução

Vamos considerar o seguinte código C que satisfaz os requisitos do enunciado:

```
int func_isa_R1(int *a, int *b, int dim)
    {
    int i;
    for (i = 0; i < dim; i++) {
        *a=b[i]-10;
```

```
            printf ("->%d", *a);
            }
        return *a;
        }
```

Vamos começar por identificar os registos que iremos usar e respectiva finalidade (objectivo: facilitar a escrita/análise do código e saber quais os registos que deverão ser salvaguardados por esta função enquanto "chamada" e enquanto "chamadora" do `printf`):

Reg	Variável
%eax	a
%edi	*a
%edx	*b
%esi	dim
%ebx	i

Vamos agora construir o código do **corpo** da função e, dentro deste, começar pelo código no interior do ciclo.

Código correspondente a `*a=b[i]-10;` :

```
        movl    12(%ebp),%edx
        movl    (%edx,%ebx,4),%eax
        subl    $10,%eax
        movl    %eax,(%edi)
```

Código correspondente a `printf ("->%d", *a);`:

```
         ??                  ; passagem do 1º argumento para printf
        pushl   %eax
        call    _printf
```

Código genérico em C associado ao ciclo `for` :

```
        expr_inic ;
        cond = expr_test ;
        if (! cond)
            goto done;
    loop:
        body_statement
        act_expr ;
        cond = expr_test ;
        if (cond)
            goto loop;
    done:
```

Código em C para o ciclo `for` neste exercício concreto (em que a "variável" `cond` foi sempre substituída por `expr_test`), e respectiva codificação em *assembly*:

```
        i = 0 ;                         xorl    %ebx,%ebx
        if (i >= dim)                   cmpl    %esi,%ebx
            goto done;                  jge     L6
    loop:                           L5:
        *a=b[i]-10;                                         ;
transcrever código feito em cima
        printf ("->%d", *a);                                ;
transcrever código feito em cima
        i++ ;                           incl    %ebx
        if (i < dim)                    cmpl    %esi,%ebx
            goto loop;                  jl      L5:
    done:                           L6:
```

Está quase concluído o código para o corpo da função. Falta apenas transferir, no início do corpo, os valores dos argumentos, ainda na *stack*, para os registos que se convencionou usar na função (na tabela, em cima), e preparar, no fim, o valor de retorno da função (para o registo `%eax`):

```
        movl    8(%ebp),%edi
        movl    16(%ebp),%esi

        movl    (%edi), %eax
```

No entanto, uma análise cuidada mostra ainda que:
- há uma instrução dentro do ciclo que está repetidamente a fazer a mesma coisa (`movl 12(%ebp),%edx`) sem que aparentemente haja alteração aos conteúdos desses registos no interior do ciclo; se calhar poderia ser retirada para antes do início do ciclo;
- por outro lado, esta função antes de chamar o `printf` deveria ter tido o cuidado de salvaguardar alguns registos (**quais?**), e depois recuperá-los quando regressasse do `printf`.

Mas... um estudo cuidado destes 2 factos poder-nos-á indicar que provavelmente não será preciso fazer nada ao código existente, que ele já estará correcto. **Porque será?**

Falta construir agora o código do **arranque** e do **término** desta função, que contemple os aspectos de gestão da *stack frame* e a salvaguarda/recuperação de registos:

- arranque:

```
        pushl   %ebp
        movl    %esp,%ebp
        pushl   %edi
        pushl   %esi
        pushl   %ebx
```

- término

```
        popl    %ebx
        popl    %esi
        popl    %edi
        popl    %ebp
        ret
```

Considere a expressão aritmética com inteiros `a=b[i]-10`, inserida no corpo de uma função em C, em que `a` é uma variável global escalar (cujo apontador foi passado como 1º argumento), `b` é um array global (cujo apontador foi passado como 2º argumento), `i` é a única variável escalar da função.

a) (...)
b) (...)
c) (...)
d) Supondo que esta era uma função-folha (i.e., que não invoca nenhuma outra), aponte as diferenças que deveria encontrar entre a estrutura da stack que mostrou em b) e a que encontraria numa arquitectura RISC de 32 bits.
*Nota: a resolução correcta desta alínea apenas é exigida para a classificação **B**.*

Sugestão de resolução

Estrutura da *stack* em b):
- de `Mem[(%ebp)+12]` a `Mem[(%ebp)+15]`, deverá estar o 2º argumento, o apontador para o

array global com 4 *bytes*;
- de `Mem[(%ebp)+8]` a `Mem[(%ebp)+11]`, deverá estar o 1º argumento, a variável `a`, do tipo `int`, com 4 *bytes*;
- de `Mem[(%ebp)+4]` a `Mem[(%ebp)+7]`, deverá estar o endereço de retorno, com 4 *bytes*;
- de `Mem[(%ebp)]` a `Mem[(%ebp)+3]`, deverá estar o valor do anterior *frame pointer*, com 4 *bytes*;
- de `Mem[(%ebp)-4]` a `Mem[(%ebp)-1]`, deverá estar o conteúdo que o registo `%esi` tinha no início da execução da função, com 4 *bytes*.

Qual a importância do facto da função ser folha ou ramo?
No IA-32, nenhuma.
Mas num processador RISC, em que normalmente o endereço de retorno é guardado num registo, (i) numa função não-folha é preciso salvaguardar na *stack* o conteúdo do registo que contém o endereço de retorno antes de chamar uma outra função, e (ii) numa função-folha tal não é necessário.

Assim, quais as diferenças na stack frame do IA-32 e dum processador RISC? Vejamos:
- argumentos: o RISC tem registos que cheguem para este caso concreto, não precisa da *stack*;
- endereço de retorno: como já se viu, o RISC não necessita da *stack*;
- salvaguarda de registos: os que existem num RISC são suficientes para que não seja preciso salvaguardar nenhum (dos reservados para a função chamadora), uma vez que esta função não vai chamar outra;
- variáveis escalares: mesmo que houvesse variáveis escalares na *stack* do IA-32, muito provavelmente não seria preciso usar a *stack* para o RISC, pois tem registos que chegue; só se houvesse variáveis estruturadas locais... mas não é este caso;
- actualização do *frame pointer*: se o RISC não necessita de *stack* para a função, para quê o *frame pointer*?

Em resumo: o processador RISC não precisa de usar a *stack* para apoio à gestão do contexto desta função!

R2.
Ao analisar código assembly gerado por um compilador de C, encontraram-se as seguintes linhas referentes a uma estrutura de controlo:

```
   movl  8(%ebp), %eax
   leal  -1(%eax), %edx
   cmpl  0,%edx
   jz    .L9
.L10
   ...
   decl  %edx
   jnz   .L10
.L9
```

Mostre duas possíveis versões do código original em C. Explicite o raciocínio efectuado.

Sugestão de resolução

Uma 1ª análise deste pedaço de código mostra 2 características pertinentes: (i) a existência de uma comparação no ínicio à qual está associada uma instrução de salto codicional - `cmp`$_{cc}$ seguido de `j`$_{cc}$, um típico teste de condição - e (ii) uma operação de decremento a anteceder um salto condicional para trás - `dec` seguido de `j`$_{cc}$, uma situação típica de teste de uma condição de saída de um ciclo.

Com base nestas pistas, se ainda não resolveu o pb, **tente resolvê-lo antes de continuar...**

Uma 2ª análise deste pedaço de código vai permitir identificar que variáveis se estariam a usar no original em C, e que tipo de variáveis seriam:

- a 1ª instrução parece carregar num registo um valor que está na memória num local que já foi referenciado várias vezes noutros exercícios como sendo o local onde normalmente se encontra o 1º argumento passado para a função; como é um valor de 4 *bytes*, será muito provavelmente um inteiro ou um apontador; as 2 instruções que se lhe seguem claramente indicam que deverá ser um inteiro com sinal; vamos então considerá-lo como o argumento `int arg_1`;
- a 2ª instrução carrega um valor no registo `%edx`; depois é feito um teste sobre ele (imediatamente a seguir) e esse valor é utilizado como variável de controlo do ciclo (mais adiante); isto sugere que o registo `%edx` é utilizado para conter o valor de uma variável local, um inteiro, a que chamaremos `var_loc`.

De notar ainda que esta 2ª instrução é um caso típico de aproveitamento da instrução `lea` pelo compilador, para efectuar uma operação do tipo `addl Src1,Src2,Dest`; neste caso concreto, a instrução `leal -1(%eax),%edx` é usada pelo compilador para substituir esta, que não existe no IA-32: `addl -1,%eax,%edx`.

Vamos então anotar/comentar o código em *assembly*:

```
   movl  8(%ebp), %eax     ; %eax= arg_1
   leal  -1(%eax), %edx    ; var_loc= arg_1-1
   cmpl  0,%edx            ; comparar var_loc com 0
   jz    .L9               ; saltar para o fim se var_loc=0
.L10                       ; início de um ciclo
   ...
```

```
   decl   %edx             ; var_loc += -1
   jnz    .L10             ; repetir o ciclo se var_loc!=0
.L9
```

A partir daqui temos a solução mais óbvia e directa do possível código original em C:

```
void exerc_R2 (int arg_1, ...)
   int var_loc;
   ...
   var_loc= arg_1-1;
   if (var_loc=0) goto done;
   do {
      ...
      var_loc += -1;
      }
   while (var_loc!=0);
 done:
```

Mas esta versão de código original em C não é elegante e é desaconselhada por usar um `goto`.
Uma melhor e mais correcta versão seria:

```
void exerc_R2 (int arg_1, ...)
   int var_loc;
   ...
   var_loc= arg_1-1;
   while (var_loc!=0)
      {
      ...
      var_loc += -1;
      }
```

R3.
A compilação de uma função em C, que recebe um único parâmetro, produziu o seguinte fragmento de código em IA32:

```
1      movl    8(%ebp),%ebx
2      xorl    %ecx,%ecx
3      cmpl    %ebx,%ecx
4      jge .L4
5  .L6:
6      incl    %ecx
7      cmpl    %ebx,%ecx
8      jl .L6
9  .L4:
10     movl    %ecx,%eax
11     leave
12     ret
```

a) Anote pormenorizadamente o código IA-32 gerado.
*Nota: a resolução desta alínea é para o nível **A**.*

Sugestão de resolução

Esta sugestão de anotar o código vai escrever como comentário/anotação uma descrição que permita visualizar o código em C :

```
        1      movl    8(%ebp),%ebx        ;  %ebx=arg_1     (int)
        2      xorl    %ecx,%ecx           ;  %ecx=0         (int
var_loc=0)
        3      cmpl    %ebx,%ecx           ;  compara var_loc:arg_1
        4      jge .L4                     ;  if (var_loc >= arg_1)
goto done
        5  .L6:                            ;loop:
        6      incl    %ecx                ;  var_loc++
        7      cmpl    %ebx,%ecx           ;  compara var_loc:arg_1
        8      jl .L6                      ;  if (var_loc < arg_1) goto
loop
        9  .L4:                            ;done:
        10     movl    %ecx,%eax           ;  %eax=var_loc (valor de
retorno da func em %eax)
        11     leave                       ;  recupera
old_frame_pointer
        12     ret                         ;  return (var_loc)
```

A compilação de uma função em C, que recebe um único parâmetro, produziu o seguinte fragmento de código em IA32:

```
(...)
```

a) (...)
b) Considerando que aquele código possui um ciclo implícito, escreva em C a função original.

Sugestão de resolução

Indo do caso mais simples para o mais complexo, esta função poderia ser um ciclo `do...while`, `while`, ou `for`.
Atendendo ao teste inicial (antes do ciclo) de uma condição que é a oposta à condição de teste existente dentro do ciclo, e ainda ao facto de existir uma variável de contagem dentro do ciclo que depois é utilizada numa comparação, é razoavelmente óbvio que este é um ciclo `for`.

A função original em C deveria ser:

```
void func_R3 (int arg_1)
   int var_loc;
   ...
   for (var_loc=0, var_loc<arg_1, var_loc++)
      {
      }
   ...
   return (var_loc);
```

Avaliação de desempenho

R1
Comente as seguintes afirmações:

a) "Sempre que me pedem para melhorar o desempenho de uma dada aplicação (sem alterar o algoritmo) procedo da seguinte maneira e por esta ordem: (i) olho para o código e procuro todos os ciclos ("loops"), (ii) aplico todas as técnicas conhecidas de optimização de código, incluindo loop unroll e execução paralela de operações sobre elementos de arrays, (iii) e recomendo que o cliente troque o seu CPU por outro com frequência de clock superior"

Sugestão de resolução

Primeira questão a considerar é a ordem das acções a tomar: parece acertada neste caso, embora seja cedo para comentar se é suficiente ou não.

Segunda questão: cada uma das acções estará correcta e será adequada? Vamos analisar caso a caso:

(i) A código da aplicação poderá conter muitas funções e cada com muitos ciclos. Mas será que a utilização típica desta aplicação utiliza da mesma maneira todas as funções e ciclos? Algumas destas funções serão muito provavelmente para resolverem situações de excepção (e não a regra), pelo que o desempenho destas funções poderá não ser crítico. Outras funções apenas serão utilizadas uma vez (funções de inicialização de *arrays*, por exemplo) ou muito esporadicamente. A 1ª acção a tomar seria então tentar identificar as partes do código onde o CPU necessita de mais tempo para as executar - usando um analisador de perfis, ou *profiler* - seleccionando deste modo quais as partes do código onde valeria a pena investir tempo para melhorar o desempenho (tornando, eventualmente, o código menos legível, e dificultando a sua manutenção futura).

(ii) Uma vez identificadas as partes do código onde vale a pena investir na optimização, conviria começar por analisar o código gerado pelo compilador com um nível de optimização aceitável, para identificar quais as optimizações que o compilador já efectuou automaticamente, e quais as que não pode introduzir, e porquê. Só então, valerá a pena procurar introduzir modificações na codificação que permitam ultrapassar as limitações do compilador, garantindo contudo que o resultado final continua correcto. Mas atenção: há modificações que dependem da arquitectura do processador onde o código vai ser executado, fazendo com que algumas modificações melhorem o desempenho numas arquitecturas, mas piorem noutras. Estas modificações, designadas por optimizações dependentes da máquina (e que incluem as relacionadas com *loop unroll* e introdução de paralelismo), não deverão ser introduzidas de ânimo leve; deverão ser comprovadas/validadas na arquitectura alvo onde a aplicação irá ser executada.

(iii) Se após a introdução das melhorias referidas anteriormente o resultado final for ainda insatisfatório, então poderá valer a pena propor alterações à arquitectura do equipamento do cliente; mas se as melhorias introduzidas forem suficientes, não parece ser eticamente correcto justificar *up-grades* de equipamento com base em necessidade de melhoria do desempenho desta aplicação. E quanto a *up-grades*, a opção de melhoria da frequência do *clock* é apenas uma entre várias, e normalmente é a que menor impacto tem no desempenho; provavelmente mais crítico poderá ser analisar a quantidade de memória existente nos níveis superiores da hierarquia de memória e, se insuficientes, aumentar a quantidade de memória onde necessário. Mas um estudo sério para quantificar estas necessidades ultrapassa o âmbito desta disciplina...

Comente as seguintes afirmações:

a) ...

b) "Quando se tenta optimizar o desempenho na execução de uma aplicação, a opção de loop unroll num ciclo com muitas iterações deve ser evitado, senão, ao desenrolar por completo o ciclo, arrisco-me a ficar com um pedaço de código muito grande que não cabe todo na cache, e portanto sou fortemente penalizado com excessivos acessos à memória com cache miss"

Sugestão de resolução

Esta afirmação contém vários factos, uns correctos, outros não. A questão aqui é identificar onde estão as falhas no raciocínio exposto.

À primeira vista, não parece fazer sentido excluir a hipótese de se usar o *loop unroll* para melhorar o desempenho... A justificação apresentada - "*arrisco-me a ficar com um pedaço de código muito grande que não cabe todo na cache*" - tem lógica, mas parte de um pressuposto errado: o de que a operação de *loop unroll* significa "*desenrolar por completo o ciclo*".
O desenrolar de um ciclo não implica fazê-lo por completo, mas tão somente diminuir o nº de iterações do ciclo; a simples duplicação das operações a efectuar no interior de um ciclo, conduz a uma redução para metade do nº de iterações do mesmo ciclo; e o triplicar das operações, uma redução para 1/3. Assim, se se proceder por passos, é possível introduzir modificações significativas sem se ser penalizado pelo facto de a totalidade do código do ciclo não caber na *cache*.
Outros pb's surgiriam provavelmente antes de se colocar esta questão da *cache*, como por ex. a insuficiência do nº de registos para permitir replicações das operações no interior do ciclo, o que iria requerer o uso da *stack* em memória (para armazenar resultados intermédios) e respectivas consequências em tempos de execução mais longos.
E há ainda a considerar o facto de o desempenho máximo estar limitado à latência das operações que são necessárias executar no interior de cada ciclo, pelo que, se se conseguir obter um desempenho próximo desse máximo teórico sem desenrolar o ciclo, é pouco provável que o desenrolar do ciclo contribua para melhorar ainda mais o desempenho.

A última parte da afirmação, a de se ser "*fortemente penalizado com excessivos acessos à memória com cache miss*" está integralmente correcta. Com efeito, se não for possível colocar o código todo do ciclo na *cache*, isso significaria que, de cada vez que se executasse uma iteração do ciclo, iria ocorrer um ou vários *cache misses* com as consequentes penalizações (*cache penalties*) devido à maior latência no acesso ao nível mais baixo da hierarquia de memória.

R/B1

A seguinte função calcula a soma dos elementos numa lista ligada:

```
1 int soma_lista(list_ptr ls)
2 {
3     int soma=0;
4
5    for (; ls; ls=ls->next)
6         soma += ls->data;
7     return soma
8 }
```

O código assembly que um compilador geraria para este ciclo, e respectiva conversão para operações do P6 (IA32), teria o seguinte aspecto:

Instruções em *assembly*	Operações da *Execution Unit* do P6

`.L43:`
`addl 4(%edx),%eax`
`movl (%edx),%edx`
`testl %edx,%edx`
`jne .L43`
	`load 4(%edx,0)`
	→ `t.1`
	`addl t.1,%eax.0`
	→ `%eax.1`
	`load (%edx.0)`
	→ `%edx.1`
	`testl %edx.1,%edx.1`
	→ `cc.1`
	`jne-taken cc.1`

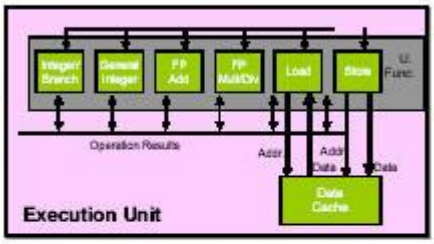
Recordar que:
- a unidade de execução do P6 tem várias unidades funcionais, sendo apenas algumas replicadas;
- a latênca dessas unidades não é igual para todas elas, e que algumas permitem execução encadeada no seu interior;
- o P6 permite a execução de instruções fora de ordem e que pode, em cada instante, enviar simultaneamente 3 instruções para a unidade de execução (i.e., é superescalar nível 3, ou em terminologia inglesa, 3-way superscalar).

a) Desenhe um diagrama que mostre a sequência de operações para as primeiras 3 iterações do ciclo (use um tipo de diagrama semelhante ao utilizado nos slides das aulas).

Analisemos a lista que foi apresentada no enunciado para "*Recordar* ".
Quais as implicações para a resolução deste exercício?
- se o P6 tem várias unidades funcionais que podem ser replicadas, importa ver quais são, quantas réplicas existem (slide nº 8 de AvDes_2 replicado em baixo) e quais são as necessidades do código a executar: o código tem 2 instruções aritméticas/lógicas sobre inteiros, 2 de *load* e 1 de salto; o P6 tem 2 unidades aritméticas, sendo uma delas tb de saltos, e 1 de *load*; mas a 1ª instrução produz dados para a 2ª, por isso estas não podem estar em paralelo, mas devem ser executadas uma após outra; idênticas dependências de dados encontram-se entre a instrução seguinte de *load* e o teste, e entre o teste e o salto; e uma análise de dependências irá ter de ser tb feita para as sucessivas iterações do ciclo;

Algumas potencialidades do Intel P6

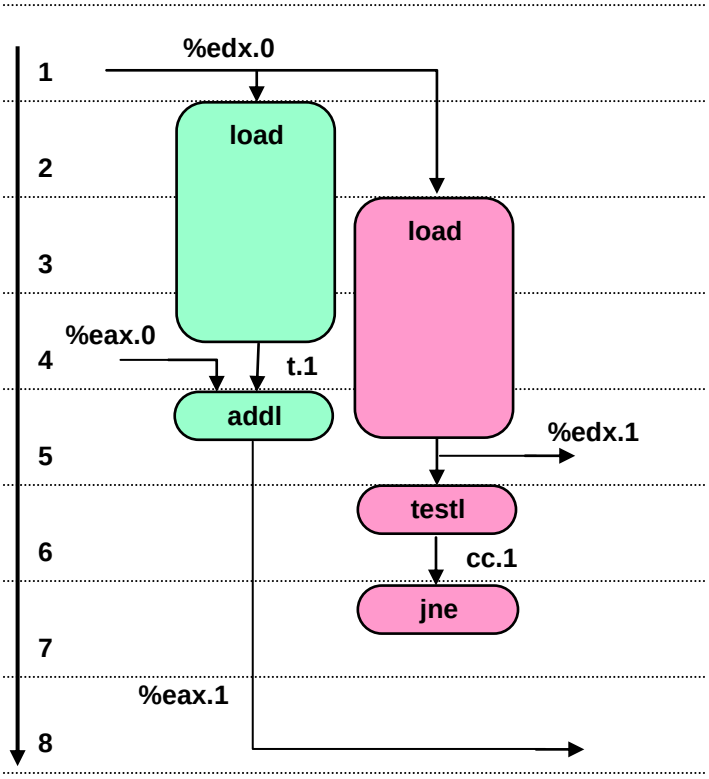
- **Execução paralela de várias instruções**
  - 2 integer (1 pode ser branch)
  - 1 FP Add
  - 1 FP Multiply ou Divide
  - 1 load
  - 1 store



- Algumas instruções requerem > 1 ciclo, mas podem ser encadeadas

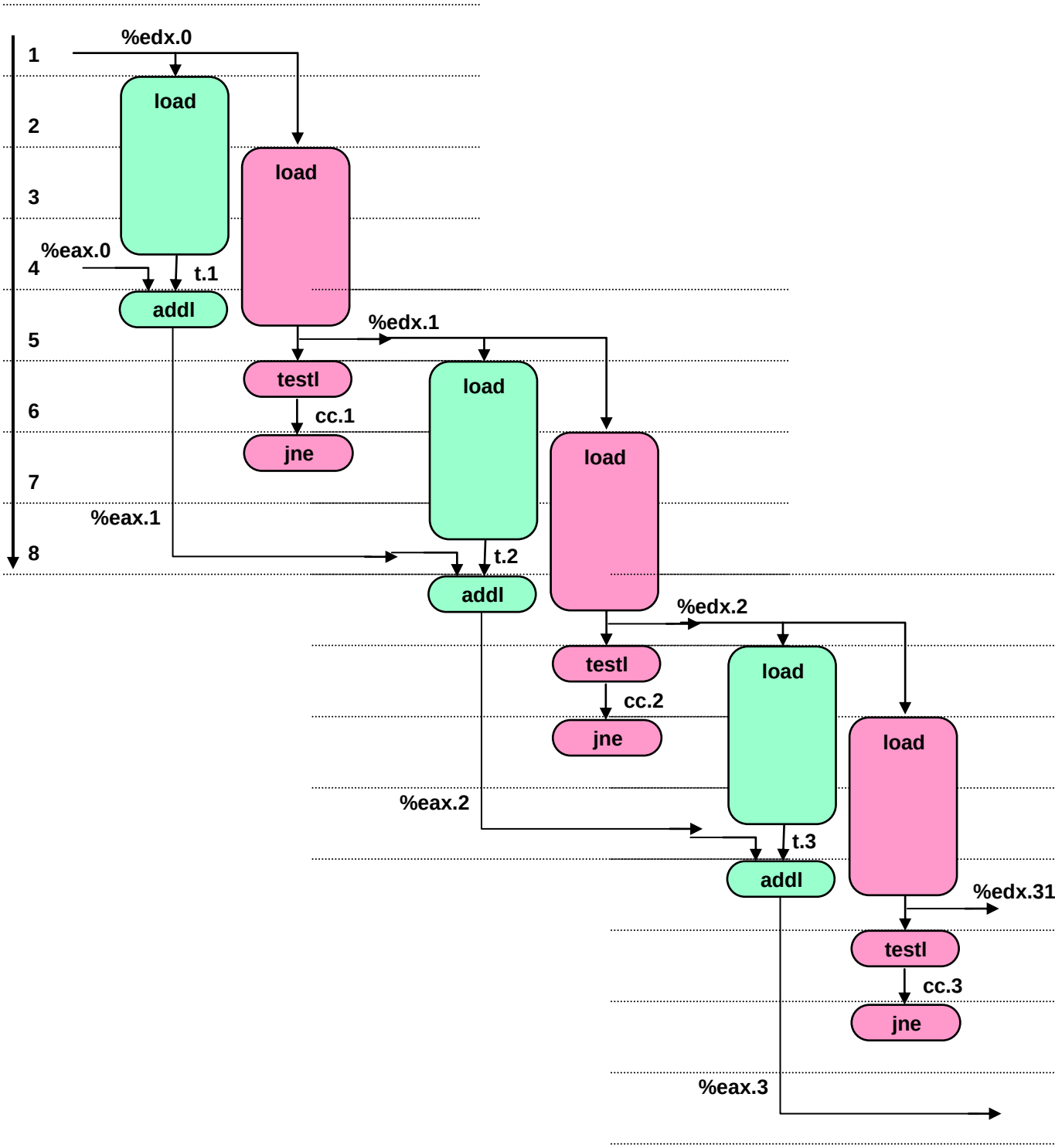
Instrução	Latência	Ciclos/Emissão
Load / Store	3	1
Integer Multiply	4	1
Integer Divide	36	36
Double/Single FP Multiply	5	2
Double/Single FP Add	3	1
Double/Single FP Divide	38	38

- as únicas instruções neste código que têm latência com duração superior a 1 ciclo de *clock* são as de *load* (3 ciclos, e encadeados, i.e., a unidade de *load* aceita processar novos dados em cada ciclo de *clock*, embora necessite de 3 ciclos para concluir a operação de *load*);
- se o P6 admite execução de instruções fora de ordem, então isto significa que é possível iniciar a execução de uma instrução antes da anterior ter terminado, ou, ainda melhor, antes de a anterior sequer ter iniciado; e sendo superescalar nível 3 (e não nível 2 como referido, por lapso, nas aulas), significa ainda que pode começar a execução simultânea de até 3 operações na *Execution Unit*, desde que haja unidades funcionais disponíveis; analisando agora o código deste pb: a 1ª instrução de *load* e a 1ª a iniciar; a 2ª terá de aguardar que a 1ª termine; a 3ª, que não depende das anteriores, poderia começar ao mesmo tempo que a 1ª, se houvesse 2 unidades de *load*; como não há, e como a unidade de *load* está disponível no ciclo de *clock* seguinte, é aí que começa; a 4ª só pode começar depois da 3ª terminar, e a 5ª depois da 4ª; há aqui nitidamente apenas 2 conjuntos de instruções que podem ser executados em paralelo (a cores diferentes na figura), embora ambos dependam da mesma variável no início do ciclo de iteração:



O diagrama representa a execução de um ciclo da iteração (excluindo a parte da busca e descodificação da instrução, que tb estão incluidos no *pipeline* global), ao longo de 7 ciclos de *clock*. De notar que o próximo ciclo da iteração apenas poderá iniciar 4 ciclos de *clock* após este ciclo de iteração ter iniciado (que é quando o novo apontador para o início da lista, em %edx, se torna disponível).

Se agora juntarmos os diagramas dos 3 primeiros ciclos da iteração, validando se em cada ciclo de *clock* todos os recursos necessários estão disponíveis, então teremos o seguinte diagrama em resposta ao solicitado na alínea:



A seguinte função calcula a soma dos elementos numa lista ligada:

```
1 int soma_lista(list_ptr ls)
2 {
3     int soma=0;
4
5    for (; ls; ls=ls->next)
6         soma += ls->data;
```

```
7      return soma
8  }
```

a) ...
b) As medições efectuadas para este ciclo mostraram um valor de CPE de 4.00. Este valor está consistente com o diagrama desenhado na alínea anterior?

De acordo com o diagrama e com o que ficou escrito atrás, cada nova iteração apenas se pode iniciar após 4 ciclos de *clock*, o que está consistente com o valor de CPE de 4.00.
Os factores que limitam o desempenho em cada iteração são **(i)** o facto de a operação que carrega da memória o apontador para o próximo valor da lista necessitar de 3 ciclos de *clock*, e **(ii)** ao qual se adiciona 1 ciclo de *clock* de espera, devido à incapacidade de o compilador verificar que se trocasse a ordem de execução das 2 operações de *load* poderia melhorar o desempenho global para um valor de CPE de 3.00.

Desafio para estudantes Excelentes: modificar o código em C para que a ordem de execução em *assembly* permita a optimização referida.

A seguinte função calcula a soma dos elementos numa lista ligada:

```
1  int soma_lista(list_ptr ls)
2  {
3      int soma=0;
4
5     for (; ls; ls=ls->next)
6          soma += ls->data;
7      return soma
8  }
```

a) ...
b) ...
c) Considere agora que a arquitectura do P6 não era superescalar, i.e., em cada ciclo de clock apenas poderia ter início uma nova instrução na execution unit. O valor de CPE vai ser alterado? Redesenhe o diagrama com a sequência temporal de operações em 3 iterações e estime o novo valor de CPE.

A análise do diagrama em cima mostra que durante a 1ª iteração nenhum par de instruções começa no mesmo ciclo de *clock*; tal só acontece nos 2 primeiros ciclos de *clock* de cada iteração, por sobreposição com a iteração anterior. Isto significa que cada nova iteração terá de atrasar 2 ciclos de *clock* relativamente à iteração anterior, fazendo com que o valor de CPE suba para 6.00.

A seguinte função calcula a soma dos elementos numa lista ligada:

```
1  int soma_lista(list_ptr ls)
2  {
3      int soma=0;
4
5     for (; ls; ls=ls->next)
6          soma += ls->data;
7      return soma
8  }
```

a) ...
b) ...
c) ...
d) Considere agora uma função ligeiramente diferente, que calcula o produto dos elementos em vez da soma. Sabendo que o produto tem uma latência 4 vezes superior à adição, o valor do CPE vai também aumentar? Continue a considerar a arquitectura do P6.

A análise do diagrama em cima mostra que se substituirmos a operação de adição pela multiplicação com uma latência de 4 ciclos de *clock*, não vai haver qualquer alteração no desempenho da execução de cada iteração, uma vez que o resultado da adição tinha sempre que esperar 3 ciclos *clock* antes de ser utilizada.
Assim, o valor de CPE deverá manter-se em 4.00.