Curso: LMCC **Exame 2ª Chamada** 29/Jun/05
Disciplina: Arquitectura de Computadores **Duração:** 2h30m

Nota: Apresente sempre o raciocínio ou os cálculos que efectuar; o não cumprimento desta regra equivale à não resolução do exercício. Use o verso do enunciado do exame como papel de rascunho.

1. Responda **apenas a uma** destas questões:

a) **(R)** Imagine que é interpelado no meio da rua por um repórter de TV (do programa "2010") e que lhe pedem que dê uma explicação breve sobre **um** dos seguintes tópicos. **Escreva** o que diria (seja rigoroso!).

AGP ; BIOS ; *chipset* ; DRAM ; *flash memory* ; latência de memória ; MP3 ; ROM ; USB ; UTF-8

Nota: o público típico deste programa é constituído por jovens amantes das tecnologias de informação, em que a maioria dos conhecimentos que possuem sobre informática não foi adquirida na escola.

b) **(R/B)** Considere um computador com um CPU RISC (32 registos, mín. 4 níveis de *pipeline* - IF, ID/OF, Ex, WB - e com *clock* de 2.5GHz) e com uma hierarquia de memória em que o nível mais baixo (logo a seguir aos registos do CPU) é externo ao CPU (fisicamente localizados a 12cm do CPU) e constituído por circuitos de memória SRAM (com uma latência de 4nseg).
Calcule o nº de ciclos de clock que este computador necessitaria para executar uma simples operação de adição (desde o fim da instrução anterior), e **proponha medidas** para corrigir esta arquitectura.

2. Analisando o conteúdo de várias células de memória num PC, a partir do endereço `0x8c10280`, encontraram-se os seguintes valores (aqui representados em vários sistemas de numeração e separados por vírgulas):

$33, 107_{11}, 63, 377_{8}, 0x4d, 97, 164_{8}, 1211_{4}, 0x6d, 0_{6}, 100_{2}, 10, 2_{7}, 86_{9}, 12, 0x2a$

a) **(A)** Considere que o CPU (IA32) tinha acabado de executar `addw 0xfff8,(%ebx,%eax,2)` quando se fez essa análise das células de memória (valor nos registos `%ebx,%eax`: `0x8c10284,0x3`). Sendo esta uma instrução do *assembly* do GNU, o resultado da operação irá ficar armazenado na memória.
Indique, justificando, a localização (endereço) de todas as células que irão armazenar o resultado da operação.
Nota: repare que provavelmente a instrução estará a processar uma variável do tipo `short int`.

Nº	Nome	Turma/Grupo:

b) **(A)** **Mostre** o conteúdo de cada uma das células de memória que foram modificadas com esta adição. **Apresente os cálculos**.
Nota: se não resolveu a alínea anterior, considere que o endereço da célula onde se encontra o *byte* menos significativo da variável é `0x8c10284`.

c) **(A/R)** **Mostre** toda a informação que deverá circular no barramento de dados (por ordem cronológica) quando esta adição for efectuada, desde o fim da instrução anterior, assumindo que:
- a instrução da operação de adição tem 32 bits e é representada por (em hexadecimal): `3e 25 8f 00`;
- os barramentos são de 32 bits;
- os registos `%eip`,`%esp`, e `%ebp` contêm, respectivamente, os seguintes valores em hexadecimal: `0x8f04802`, `0x8c12004`, `0x8c12010` .

d) **(A/R)** Sabendo que as instruções antes desta operação de adição retiraram da *stack* uma *string* com as iniciais de Matemática (*Matem*) **indique, justificando**, o conteúdo do registo `%esp` durante a execução da operação de adição.

H	e	l	l	o		w	o	r	l	d	!
48	65	6c	6c	6f	20	77	6f	72	6c	64	21

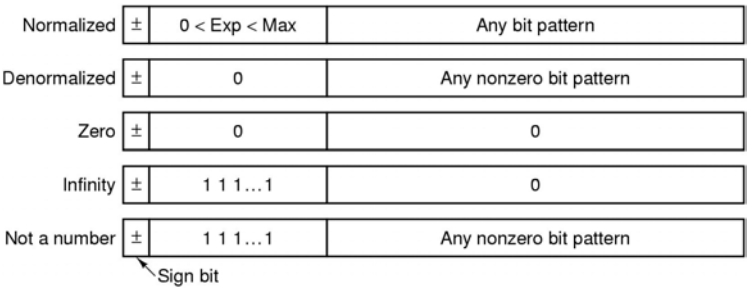
Nº	Nome	Turma/Grupo:

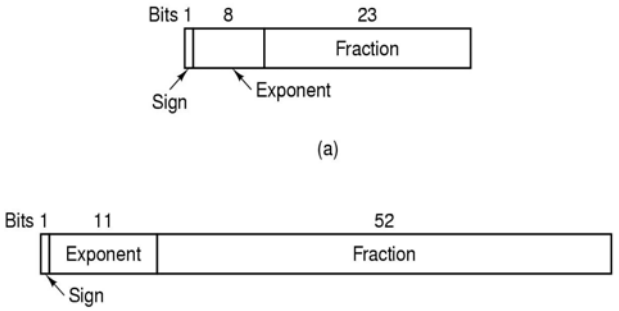
e) **(R/B)** Considere agora um **processador típico RISC de 16 bits** com as seguintes especificidades: suporte a instruções aritméticas de *add* e *mult*, bem como de *move* entre registos e memória; tem registos de `%r0` a `%r31`; permite apenas um modo de especificar um endereço de memória, através da soma de 2 registos.
Se o compilador da GNU tivessse de gerar código para esse processador, **indique** a sequência de instruções geradas para executar aquela adição (use a sintaxe do *assembler* da GNU, com as devidas adaptações para a arquitectura típica RISC)

3.

a) **(A/R)** Uma aproximação ao valor de π é considerá-lo como um fraccionário que se obtém a partir de 22/7.
Se se quisesse representar este valor como sendo um real (do tipo *float*) **mostre** como seria o valor de π representado num registo de *fp* do PC (de 64 bits).
Nota 1: Para chegar ao seu valor em binário, **deve** converter primeiro cada um dos números para binário e só depois efectuar a divisão.
Nota 2: Se não sabe dividir em binário, considere $\pi = 3.1416$

Notas de apoio (norma IEEE 754)





Valor decimal de um fp em binário:

- precisão simples, normalizado: $V = (-1)^S * (1.F) * 2^{E-127}$
- precisão simples, desnormalizado: $V = (-1)^S * (0.F) * 2^{-126}$

Nº	Nome	Turma/Grupo:

b) **(B)** Sabendo que fracção 22/7 apenas se aproxima de π nos primeiros 3 algarismos decimais, (i.e., apenas tem 3 algarismos significativos), **indique** quantos bits serão precisos na parte fraccionária de uma representação binária para manter esses 3 algarismos significativos.

c) **(R)** Considere a operação de *casting* do C, na conversão para inteiro do valor real (tipo *float*) armazenado em binário num registo *fp* com a seguinte combinação de bits: `0xf02c`. Sem efectuar qualquer operação aritmética, **mostre** qual o resultado da operação. **Apresente** o raciocínio seguido.

4. Considere as funções `combine1` e `combine2` (versões em C e *assembly* em folha anexa).

a) **(A)** **Mostre e comente** o código *assembly* que o `gcc` iria gerar (para um CPU IA32) se o ciclo `for` na função `combine2` fosse substituido por:

```
int i=0;
if (i<=length) goto feito;
do {
   x += data[i];
   i++;  }
   while (i<length);
feito:
```

Nota: Mostre apenas o código para o corpo da função após a invocação de `get_vec_start`)

Nº	Nome	Turma/Grupo:

b) **(R)** A primeira operação no corpo da função `combine2` é uma atribuição (`length=...`).
Contudo, se se analisar o código integral em *assembly* (na folha anexa apenas está o código mais pertinente do corpo da função), o `gcc` insere normalmente várias instruções antes do início do corpo da função. **Justifique** e **descreva** a funcionalidade desse código extra gerado pelo compilador.

c) **(B)** A análise do desempenho da função `combine1` num dado PC mostrou que a utilização da opção de optimização `O2` no `gcc` permitiu uma melhoria no desempenho de quase 40%. **Apresente explicações** para esta melhoria.

d) **(R)** Alterações ao código conduziram à versão de `combine2`, permitindo melhorar o desempenho 12x. **Indique** as alterações introduzidas no código C, e **mostre** como o código *assembly* gerado influenciou o desempenho da aplicação.

Nº	**Nome**	**Turma/Grupo:**

/* Código para o "combine1" */

```
void combine1(vec_ptr v, data_t *dest)
{
   int i;
   *dest =0;
   for (i = 0; i < vec_length(v); i++) {
         data_t val;
         get_vec_element(v, i, &val);
         *dest = *dest + val;
   }
}
```

```
_combine1:
     ...
     movl    12(%ebp), %esi
     movl    8(%ebp), %edi
     movl    $0, (%esi)
L45:
     movl    %edi, (%esp)
     call    _vec_length
     cmpl    %eax, %ebx
     jge     L50
     movl    %ebx, 4(%esp)
     leal    -16(%ebp), %eax
     incl    %ebx
     movl    %edi, (%esp)
     movl    %eax, 8(%esp)
     call    _get_vec_element
     movl    -16(%ebp), %eax
     addl    %eax, (%esi)
     jmp     L45
L50:
     ...
     ret
```

/* Código para o "combine2" */

```
void combine2(vec_ptr v, data_t *dest)
{
   int i;
   int length = vec_length(v);
   data_t *data = get_vec_start(v);
   data_t x =0;

   *dest =0;
   for (i = 0; i < length; i++) {
               x = x + data[i];
   }
   *dest = x;
}
```

```
_combine2:
     ...
     call    _vec_length
     ...
     call    _get_vec_start
     movl    $0, (%edi)
     xorl    %ecx, %ecx
     xorl    %edx, %edx
     cmpl    %esi, %ecx
     jge     L74
L72:
     movl    (%eax,%edx,4), %ebx
     incl    %edx
     addl    %ebx, %ecx
     cmpl    %esi, %edx
     jl      L72
L74:
     movl    %ecx, (%edi)
     ...
     ret
```

Nº	**Nome**	**Turma/Grupo:**