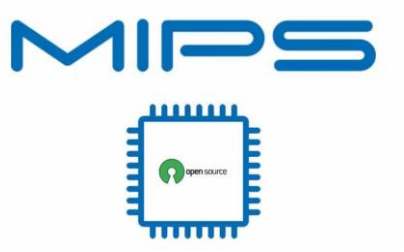
Faculdade de Computação

Arquitetura e Organização de Computadores 1

Laboratório de Programação Assembly 2

Prof. Cláudio C. Rodrigues

Programando a Arquitetura MIPS



MIPS, acrônimo para **Microprocessor without Interlocked Pipeline Stages** (microprocessador sem estágios intertravados de pipeline), é uma arquitetura de microprocessadores RISC desenvolvida pela MIPS Computer Systems.

Arquitetura

O MIPS é uma arquitetura baseada em registrador, ou seja, a CPU usa apenas registradores para realizar as suas operações aritméticas e lógicas. Existem outros tipos de processadores, tais como processadores baseados em pilha e processadores baseados em acumuladores. Processadores baseados no conjunto de instruções do MIPS estão em produção desde 1988. Ao longo do tempo foram feitas várias melhorias do conjunto de instruções. As diferentes revisões que foram introduzidas são MIPS I, MIPS II, MIPS III, MIPS IV e MIPS V. Cada revisão é um super conjunto de seus antecessores. Quando a MIPS Technologies saiu da Silicon Graphics em 1998, a definição da arquitetura foi alterada para definir um conjunto de instruções MIPS32 de 32 bits e um MIPS64 de 64 bits.

Projetos MIPS são atualmente bastante usados em muitos sistemas embarcados como dispositivos Windows CE, roteadores Cisco e consoles de games como: Nintendo 64, Playstation, Playstation 2 e Playstation Portable.

A Filosofia do Projeto do MIPS:

- A simplicidade favorece a regularidade.
  O menor é (quase sempre) mais rápido. Levando em conta que uma corrente elétrica se propaga cerca de um palmo por nanosegundo, circuitos simples são menores e portanto, mais rápidos.
- Um bom projeto demanda compromissos.
- O caso comum DEVE ser mais rápido.
  É muito melhor tornar uma instrução que é usada 90% do tempo 10% mais rápida do que fazer com que uma instrução usada em 10% das vezes torne-se 90% mais rápida. Esta regra é baseada na "Lei de Amdahl".

Instruções:

I. Apresentar as soluções usando a linguagem de montagem da Arquitetura de processadores MIPS.
II. O trabalho deve ser desenvolvido em grupo composto de 1 até 5 (um até cinco) discentes e qualquer identificação de plágio sofrerá penalização;
III. Entrega dos resultados deverá ser feita por envio de arquivo zipado com os seguintes artefatos de software: Memorial descritivo das soluções em pdf; arquivo em txt com os códigos que solucionam os problemas propostos escritos em MIPS assembly.
IV. Submeter os documentos na plataforma MS Teams, dentro do prazo definida para a atividade.

Desafios de Programação MIPS:

P1) A constante de Euler-Mascheroni (também chamada de constante de Euler) é uma constante matemática, geralmente denotada pela letra grega gama (**γ**), com múltiplas utilizações em Teoria dos números. Ela é definida como o limite da diferença entre a série harmônica (H) e o logaritmo natural (ln), $\gamma_n \approx H_n - \ln(n)$.

O limite existe e é aproximadamente ***γ ≈ 0,57721566490153286060651209008240***2.

Série harmônica: Escreva em *MIPS assembly* uma função denominada ***Harmonica(n)***, que calcule a série Harmônica para um determinado valor de **n**. Em matemática, a série harmônica é a série infinita definida como: $\sum_k^\infty \frac{1}{k} = 1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \cdots$

Desenvolva o **algoritmo da série harmônica**, realizando os cálculos com ***precisão simples*** e ***precisão dupla***. Com *precisão simples*, quando N = 10.000, a soma é precisa para 5 dígitos decimais, quando N = 1.000.000 é precisa para apenas 3 dígitos decimais, quando N = 10.000.000 é precisa para apenas 2 dígitos decimais. Na verdade, uma vez que **N** atinge 10 milhões, o somatório satura. Embora a soma harmônica divirja para o infinito, na matemática de ponto-flutuante ela converge para um número finito! Isso contraria a crença popular de que se resolver um problema que requer apenas 4 ou 5 dígitos decimais de precisão, então você está seguro usando um tipo que tem 7 dígitos de precisão. Muitos cálculos numéricos (por exemplo, integração ou soluções para equações diferenciais) envolvem somar um monte de pequenos termos. Os erros podem se acumular. Esse acúmulo de erros de arredondamento pode levar a sérios problemas.

O programa deve estar contido no arquivo "**gama.asm**".

P1. Escreva um programa em ***MIPS assembly***, que implemente a função ***exp(x)*** para calcular $\mathbf{e^x}$ por desenvolvimento em Série de Taylor desprezando termos, em grandeza, inferiores a $\mathbf{10^{-5}}$.

$$e^x = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots, \qquad -\infty < x < \infty$$

A função exponencial deve somar os termos da série até aparecer um termo cujo valor absoluto seja menor que 0.00001 (precisão). Isto é, o termo $\left|\frac{x^k}{k!}\right|$ tende a zero quando **k** tende a **+∞**. Repetir o cálculo dos termos até um **k** tal que $\left|\frac{x^k}{k!}\right| < 0.00001$.

Dica: evite calcular o valor do fatorial, calcule um termo da série usando o termo anterior.

O programa deve estar contido no arquivo "**expx.asm**".

Análise Infinitesimal:

Desenvolvimento em série de Taylor da função exponencial:

$e^x = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots + \frac{x^n}{n!}$, $-\infty < x < \infty$, convergente $\forall x \in R$

Somas parciais de ordem ***n*** e ***n+1***:

$$S_n = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots + \frac{x^n}{n!}$$

$$S_{n+1} = S_n + \frac{x^{n+1}}{(n+1)!}$$

Termos de Ordem ***n*** e ***n+1***:

$$t_n = \frac{x^n}{n!}$$

$$t_{n+1} = \frac{x^{n+1}}{(n+1)!} = t_n \frac{x}{n+1}$$

A série é convergente $\forall x \in R$, isto é:

$$\lim_{n\to\infty}\{S_n\} = e^x$$

E, nesse caso:

$$\lim_{n\to\infty}\{t_n\} = 0$$

P2. *Avaliação Polinomial*: Escreva em linguagem de montagem da arquitetura *MIPS* um programa que calcule o valor do polinômio $\mathbf{p(x)=a_0+a_1x+...+a_nx^n}$ em **k** pontos distintos (valores de **x**).
Elabore duas funções para a avaliação do polinômio em um determinado ponto x. A primeira função implementa o método tradicional, a segunda função, implementa o método de Horner para avaliação do polinômio. Cada função receberá os valores de **n** (ordem do polinômio), um vetor de coeficientes do polinômio ($\mathbf{a_0, a_1, ..., a_n}$), e o valor avaliado de **x** de **p(x)**. Dica: um polinômio de ordem **n** pode ser representado pelos coeficientes guardados em um vetor de tamanho **n+1** elementos.

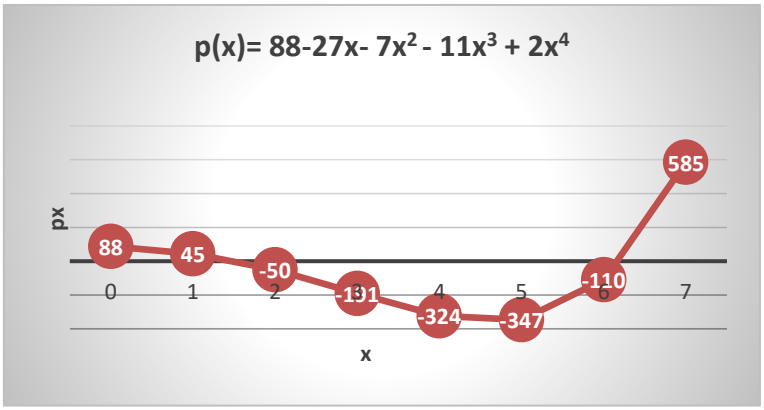
Método tradicional	**Método de Horner**
$P(x) = 2x^4 - 11x^3 - 7x^2 -27x + 88$	**$P(x) = ((((2x - 11)x -7)x - 27)x + 88)$**

Método tradicional

```
// p(x) = a[0]x(n-1) + a[1]x(n-2) + .. + a[n-1]
int tradicional(int a[], int n, int x)
{
   int poly = 0];

   // Evaluate value of polynomial
   for(i=0;i<=n;i++)
      poly += a[i]*pow(x,n-1-i);

   return poly;
}
```

Método de Horner

```
// p(x) = a[0]x(n-1) + a[1]x(n-2) + .. + a[n-1]
int horner(int a[], int n, int x)
{
   int poly = a[0];

   // Evaluate value of polynomial
   for (int i=1; i<n; i++)
      poly = poly*x + a[i];

   return poly;
}
```



O programa deve estar contido no arquivo "**polinomio1.asm**" e "**polinomio2.asm**".

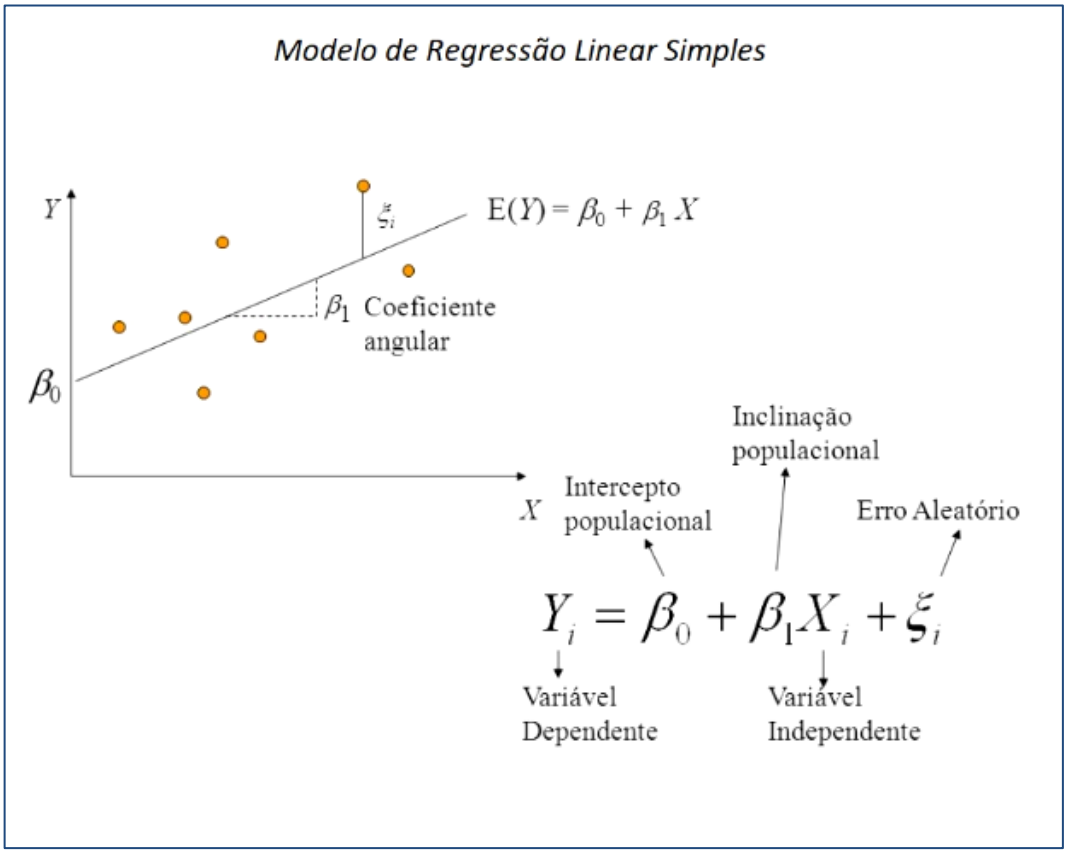
P3. Dado uma coleção de **n** pontos no plano cartesiano de coordenadas ($\mathbf{x_i,y_i}$), o método de regressão linear permite determinar os coeficientes da reta **Y = A.X+B** que se aproxima melhor do conjunto de pontos fornecidos; os coeficientes são determinados por

$$A = \frac{nS_{xy} - S_x S_y}{nS_{xx} - (S_x)^2}, \quad B = \frac{S_y - AS_x}{n}$$

Onde $S_x = \sum_{i=1}^{n} x_i$, $S_y = \sum_{i=1}^{n} y_i$, $S_{xx} = \sum_{i=1}^{n} x_i \, . \, x_i$ e $S_{xy} = \sum_{i=1}^{n} x_i \, . \, y_i$

Escreva em MIPS assembly função denominada ***regressao_linear*** que receba uma coleção de pontos (vetor de pontos) e a quantidade de pontos e retorna os coeficientes ***A*** e ***B***. Construa a função *main()* para testar a função ***regressao_linear*** e apresentar os resultados na tela de saída. Para testar o algoritmo defina o valor de **n** (>=10), gere a coleção de pontos de forma aleatória.

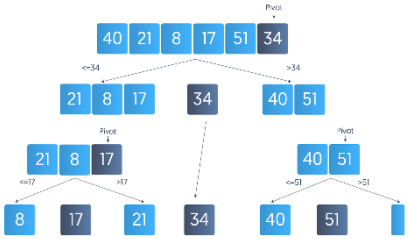
O programa deve estar contido no arquivo "**regressao.asm**".



P4. Escreva em *MIPS assembly* uma função denominada ***quicksort***, que ordena um array recursivamente. Traduza a função a seguir, em código MIPS Assembly. Escreva uma função principal para chamar e testar a função *quicksort*.

```
void quicksort(int* array, int low, int high){
    int i = low, j = high;                          // low and high index
    int pivot = array[(low+high)/2];                // pivot = middle value
    while (i <= j) {
        while (array[i] < pivot) i++;
        while (array[j] > pivot) j--;
        if (i <= j) {
            int temp=array[i];
            array[i]=array[j];                      // swap array[i]
            array[j]=temp;                          // with array[j]
            i++;
            j--;
        }
    }
    if (low < j) quick_sort(array, low, j);         // Recursive call 1
    if (i < high) quick_sort(array, i, high);       // Recursive call 2
}
```

O programa deve estar contido no arquivo "**quicksort.asm**".



P5. Em álgebra linear, uma matriz é uma tabela retangular de números reais ou complexos. Dada uma matriz A, usamos a notação Aij para representar um elemento na linha i e na coluna j. Podemos implementar uma matriz usando um array bidimensional, organizados em memória por linha (*row-major order*). Começamos a indexação em 0 para estar em conformidade com as convenções de indexação.

Multiplicação da matriz. O produto de duas matrizes $\mathbf{A}_{nxn}$ e $\mathbf{B}_{nxn}$ é uma matriz $\mathbf{C}_{nxn}$ definida por

$$C_{ij} = \sum_{k} A_{ik}.B_{kj}$$

Os fragmentos de código abaixo, descrevem o algoritmo tradicional de multiplicação de matrizes, C = AB, considerando duas estratégias de indexação **ijk** ou **ikj**.

	mulmat_ijk:	mulmat_ikj:

	`for (int i = 0; i < N; i++)`
	`  for (int j = 0; j < N; j++)`
	`  soma = 0.0;`
	`  for (int k = 0; k < N; k++)`
	`  soma += A[i][k] * B[k][j];`
	`  C[i][j] = soma;`
		`for (int i = 0; i < N; i++)`
		`  for (int k = 0; k < N; k++)`
		`  soma = 0.0;`
		`  for (int j = 0; j < N; j++)`
		`  soma += A[i][k] * B[k][j];`
		`  C[i][j] = soma;`

A sequência de referências a memória pode ter um enorme impacto no armazenamento em cache e no desempenho do algoritmo.

É apropriado considerar que, a operação de multiplicação de matrizes é um gargalo computacional para muitas aplicações. Podemos armazenar em cache explicitamente certas linhas e colunas para tornar a computação mais rápida.

Escreva em *MIPS assembly* as funções de multiplicação de matrizes (***mulmat_ijk*** e ***mulmat_ikj***). Faça uma análise de desempenho em relação ao tempo de acesso a memória, capturando as estatísticas de ***hits*** e ***misses*** nos módulos caches. Considere um cache de tamanho 1K palavras e bloco de tamanho 8 palavras, e matrizes de dimensão 512 × 512. As matrizes são armazenadas em *row-major order*. Execute a análise de ***cache miss*** para as estratégias **ijk** e **ikj** de multiplicação de matrizes, considerando caches mapeados diretamente.

O programa deve estar contido no arquivo "**mulmat-ijk.asm**".