MEMORIAL DESCRITIVO
LABORATÓRIO DE PROGRAMAÇÃO ASSEMBLY 1

AMANDA DUARTE GARCIA - 12221BCC031
CAIKE CESAR MOTA DE ARAUJO - 12221BCC030
MATHEUS FIOD SALIBA - 12221BCC024
MURILO ALVES BEPPLER - 12221BCC037

P1. POTENCIAÇÃO OU EXPONENCIAÇÃO

Para o Fragmento 1, foram declaradas três variáveis, sendo elas **x**, **n** e **result**, as quais representam a base, o expoente e o resultado, respectivamente. A partir disso, temos as seguintes funções:

1) **expo1**
   Move cada um dos valores (**x**, **n**, **result**) para seus respectivos registradores;
2) **teste**
   Verifica o valor de **n**, de modo que, se **n** for menor que zero, o resultado é imprimido na tela e o programa é finalizado, senão, a função while é chamada;
3) **while**
   Realiza o cálculo da potenciação.

Quanto ao desempenho do algoritmo, o loop while executa, no máximo, n iterações, onde n é o valor do expoente. Dentro do loop, há operações de multiplicação e subtração, e cada iteração executa em tempo constante, mas há n iterações no pior caso. Portanto, a complexidade de tempo é O(n).

Para o Fragmento 2, as mesmas variáveis foram declaradas (**x**, **n**, **result**), entretanto as funções mudaram:

1) **expo2**
   Move cada um dos valores (**x**, **n**, **result**) para seus respectivos registradores;
2) **loop**
   Verifica o valor de **n**, de modo que se **n** for igual a zero, o loop é encerrado, senão, é verificado se **x** é par, para chamar a função par, ou ímpar, para chamar a função ímpar;
3) **par**
   Eleva **x** ao quadrado e faz em **n** um deslocamento lógico para a direita, antes de voltar para a função loop;
4) **ímpar**
   Multiplica **result** por **x** e eleva **x** ao quadrado, fazendo um deslocamento lógico para a direita em **n** antes de voltar para a função loop;
5) **fimloop**
   Imprime o resultado da exponenciação na tela e encerra o programa.

Quando ao desempenho do algoritmo, o programa reduz o número total de iterações a partir de uma análise do expoente. No caso médio, o número de iterações é significativamente menor que n, tendo complexidade $O(\log_2 n)$. No pior caso, o número de iterações é limitado por n, entretanto ainda é mais eficiente que o algoritmo do Fragmento 1.

P2. AVALIAÇÃO POLINOMIAL

Para este problema, foram declaradas as variáveis **coef**, **x**, **n** e **result**, as quais representam, respectivamente, o vetor de coeficientes, o valor de x, a ordem do polinômio e o resultado do cálculo. Além disso, há as seguintes funções:

1) **main**
   Carrega as variáveis **n** e **x** em seus respectivos registradores, chama a função **tradicional** e recebe o resultado obtido dela, imprimindo-o na tela. Depois, recarrega as mesmas variáveis de antes e chama, dessa vez, a função **horner**, recebendo o resultado obtido e também o imprimindo na tela. Por fim, encerra o programa;
2) **pow**
   Função responsável por calcular potências, mantendo o loop da função **loop_pow** funcionando até que **n** seja zero;
3) **loop_pow**
   Multiplica o resultado pelo coeficiente enquanto **n** é diferente de zero;
4) **fim_pow**
   Retorna o resultado final da potenciação para a função **main**;
5) **tradicional**
   Preserva os registros dos endereços de retorno da função **pow** antes de configurar os registradores para o loop da função **loop_trad**;
6) **loop_trad**
   Com o auxílio da função **pow**, faz o cálculo de P(x) a partir do método tradicional (*poly += a[i] * pow(x, n-1-i)*) por meio de um loop até que o contador auxiliar seja igual ao valor de **n**;
7) **fim_trad**
   Retorna o resultado obtido por meio do método tradicional para a **main** e faz o tratamento dos registros;
8) **horner**
   Inicializa o resultado como o primeiro coeficiente e define um contador para a função **loop_horner**;
9) **loop_horner**
   Faz o cálculo de P(x) a partir da expressão dada a partir do método de Horner, por meio de um loop até que o contador auxiliar tenha o mesmo valor de **n**;
10) **fim_horner**
    Retorna o resultado obtido por meio do método de Horner para a **main**.

P3. RAÍZES DE SEGUNDA ORDEM

Para determinar o valor de x de um polinômio de segundo grau, inicialmente foi definida uma variável chamada **cte4** como 4 e, em seguida, diversas funções para a realização desse cálculo, levando em consideração a Regra da Tumba para evitar o cancelamento catastrófico e o tratamento para raízes com valores complexos.

1) **main**
   Faz a leitura dos coeficientes a, b e c de um polinômio de segundo grau a partir do dispositivo padrão de entrada (teclado). Em seguida, é feito um tratamento do coeficiente **a**, pois, se **a** for 0, o programa é encerrado, senão, o cálculo continua, calculando o valor de **delta**. Se **delta** < 0, há um jump para a função **complexo**, senão, calcula-se a raiz de **delta** e faz a Regra da Tumba. Se $\mathbf{b^2} > \mathbf{ac}$ e **b** > 0, a R1 será calculada de forma segura e a R2 de forma tradicional (função **primeiro_metodo**) senão, a R1 será calculada de forma tradicional e a R2 de forma segura (função **segundo_metodo**);
2) **segundo_metodo**
   Calcula R1 de forma tradicional e R2 de forma segura;
3) **primeiro_metodo**
   Calcula R1 de forma segura e R2 de forma tradicional;
4) **imprimir_resultado**
   Imprime na tela os valores de R1 e R2;
5) **complexo**
   Faz o cálculo das raízes pelo jeito tradicional da fórmula de Bhaskara levando em consideração que o **delta** é menor que zero e, consequentemente, às raízes são complexas;
6) **a_zero**
   Trata o erro de quando **a** é zero, imprimindo na tela um aviso para o usuário;
7) **fim**
   Encerra o programa.

- O programa utiliza as funções **read_float** (lê um float do teclado), **print_str** (imprime uma string na tela) e **print_float** (imprime um float na tela), definidas no arquivo *macros.asm*.

P4. PROGRAM CHALLENGE

Para este problema, foram declaradas uma matriz quadrada **M** e sua dimensão **n**. A partir disso, existem diversas funções, feitas com base no exemplo em linguagem C, com o objetivo de calcular a soma de cada linha, coluna e diagonal, comparando os resultados obtidos para determinar se dada matriz quadrada é um quadrado mágico.

1) **main**
   Utilizada para a definição de registradores para cada uma das variáveis necessárias, sendo elas a matriz **M** e sua dimensão **n**, **soma1**, **soma2**, **soma3**, a flag **magico** e o índice **i**;
2) **diagonalPrincipal**
   Realiza a soma dos elementos na diagonal principal da matriz quadrada;
3) **diagonalSecundaria**
   Realiza a soma dos elementos na diagonal secundária da matriz quadrada e, depois, compara o resultado com o valor obtido na função **diagonalPrincipal**. Se as somas forem diferentes, a análise é interrompida e o programa é encerrado, senão a análise continua;
4) **verificacao**
   Inicializa as variáveis necessárias para as próximas comparações, sendo elas **soma2**, **soma3** e o índice **j**;
5) **somas**
   Compara as somas obtidas em cada linha e as obtidas em cada coluna. Caso alguma soma tenha resultado diferente, a análise é interrompida e o programa é encerrado, do contrário, a análise continua até que todas as linhas e colunas sejam analisadas;
6) **falseMagico**
   Se o programa identifica duas somas diferentes, a flag **magico** = 0 indica que a matriz analisada não forma um quadrado mágico;
7) **trueMagico**
   Com todas as somas iguais, a flag **magico** = 1 indica que a matriz analisada forma um quadrado mágico.

- O programa utiliza a função **print_str** (imprime uma string na tela), definida no arquivo *macros.asm*.

P5. FUNÇÃO EULER PHI

Para calcular a função Phi de Euler, foram definidas três variáveis inicialmente, sendo elas **n**, **result** e **msg**, para guardar o valor de entrada, o resultado da função e uma mensagem para exibir o resultado, respectivamente. Depois disso, foram definidas as seguintes funções:

1) **main**
   Carrega as variáveis em seus respectivos registradores, definindo também um **contador**;
2) **loop**
   Chama a função **mdc** até que **n** seja igual ao **contador**;
3) **incrementar**
   Incrementa o **resultado** e o **contador**, chamando a função **loop** no final;
4) **fim**
   Imprime o resultado na tela e encerra o programa;
5) **mdc**
   Passa o **resultado** e o **contador** para registradores temporários antes de iniciar os cálculos de MDC, os quais são feitos na função **mdc_loop**;
6) **mdc_loop**
   Faz o cálculo do MDC até que o **contador** seja zero;
7) **mdc_fim**
   Encerra o cálculo do MDC e retorna o **resultado**.

MACROS.ASM

A fim de facilitar o desenvolvimento do código e corrigir erros obtidos durante a programação de alguns exercícios, foi criado um arquivo de macros com as seguintes funções definidas nele:

1) **print_str**
   Imprime na tela uma string;
2) **print_float**
   Imprime na tela um float;
3) **read_float**
   Faz a leitura de um float a partir do teclado.