7.12 EXERCÍCIOS PROPOSTOS

1. Com relação à compilação e execução de um programa no computador, responda:
   a) Quais as restrições que o processador possui em relação à execução de um programa?
   Resposta: O processador possui algumas restrições em relação à execução de um programa, tais como:
      - Arquitetura: O programa deve estar compilado para a arquitetura específica do processador. Cada processador possui uma arquitetura e conjunto de instruções próprio, e um programa compilado para um processador não pode ser executado diretamente em outro com arquitetura diferente.
      - Memória: O processador só consegue executar programas em linguagem de máquina que estejam carregados na memória e com todas as rotinas da bibliotecas incorporadas de forma estática ou dinâmica ao código executável.

   b) Quais os passos e programas necessários para a compilação e execução de um programa no computador?
   Resposta:
      i. Primeiro o compilador, que é específico para cada linguagem de alto nível, faz a tradução para a linguagem de montagem específica do processador utilizado no computador. Mas a linguagem de montagem (assembly language em inglês) é apenas uma forma intermediária de representação em baixo nível do programa, que ainda pode ser lida, compreendida e modificada pelo programador.
      ii. No passo seguinte, o programa é traduzido para a linguagem de máquina pelo montador (assembler em inglês). Normalmente, para cada tipo de arquitetura/sistema operacional, existe apenas um programa montador que realiza essa tradução para todos os compiladores de alto nível instalados no computador. O arquivo resultante está em linguagem de máquina, em formato binário, compreensível pelo processador, sendo chamado agora de programa objeto.
      iii. Contudo, esse programa objeto ainda não está pronto para ser executado. Falta ainda adicionar um conjunto de rotinas pré-definidas nas bibliotecas da linguagem de alto nível, como por exemplo o *printf* na linguagem "C". Como esse conjunto de rotinas é muito extenso, e a inclusão de todas as rotinas tornaria o tamanho final do programa executável demasiadamente grande, elas só são incorporadas ao código do programa executável quando houver alguma referência explícita a alguma dessas bibliotecas.
      iv. Essa incorporação pode se dar ainda de duas formas: utilizando-se bibliotecas estáticas ou dinâmicas. No caso das bibliotecas estáticas, as rotinas são incorporadas definitivamente ao código executável pelo programa ligador (linker em inglês), que por essa razão passa a ter um tamanho maior que o programa objeto. Se todos os programas executáveis utilizarem essa estratégia, pode haver em algum momento diversos programas carregados na memória do computador com as mesmas rotinas repetidas, resultando em um gasto adicional e desnecessário de memória.
      v. Para evitar que isso aconteça, é possível a utilização das bibliotecas dinâmicas, onde o código das rotinas é carregado apenas uma vez na memória e compartilhado por

todos os outros programas que possam necessitar delas. Mas essa verificação é feita pelo carregador (loader em inglês) no momento que o programa executável é carregado na memória para execução pelo processador. Note que o processador só consegue executar programas em linguagem de máquina que estejam carregados na memória e com todas as rotinas da bibliotecas incorporadas de forma estática ou dinâmica ao código executável.

c) Qual a diferença entre as bibliotecas estáticas e dinâmicas?
Resposta:
- As bibliotecas estáticas são vinculadas diretamente no executável do programa em tempo de compilação. Isso significa que todas as funções da biblioteca são copiadas para o executável, tornando-o maior. No entanto, a vantagem é que o programa não requer a presença da biblioteca no sistema em que será executado.
- As bibliotecas dinâmicas são vinculadas ao programa em tempo de execução. O código das funções da biblioteca é mantido em arquivos separados (com extensão .dll no Windows ou .so no Linux), e o programa usa essas bibliotecas em tempo de execução, quando necessário. Isso torna o executável menor, pois não inclui todo o código da biblioteca. No entanto, o programa depende da presença das bibliotecas dinâmicas na versão correta no sistema em que será executado. Outra vantagem é que várias aplicações podem compartilhar as mesmas bibliotecas dinâmicas, economizando espaço em memória.

2. Nos programas em linguagem de montagem, qual a finalidade das pseudo-instruções?
Resposta: As pseudo-instruções, também conhecidas como diretivas, são comandos especiais utilizados em programas escritos em linguagem de montagem para auxiliar na montagem do código-fonte, mas não são instruções executáveis pelo processador. Elas fornecem informações e instruções para o montador (assembler), que é o programa responsável por traduzir o código em linguagem de montagem para código de máquina. As pseudo-instruções têm várias finalidades, incluindo:
a) Definir constantes.
b) Reservar espaço na memória para as variáveis, com ou sem valor inicial.
c) Definir rótulos para marcar pontos específicos no programa, indicando o início de um bloco de código ou para referenciar determinados endereços de memória.
d) Informar ao montador sobre o formato do código de máquina a ser gerado, o local onde o código deve ser colocado na memória, entre outras informações de montagem.
e) Controlar aspectos do processo de montagem, como a inclusão de outros arquivos de código-fonte, a geração de listagens de montagem detalhadas e a ativação ou desativação de recursos específicos do montador.

3. Descreva resumidamente os passos envolvidos na chamada e o retorno de uma subrotina.
Resposta: A chamada e o retorno de uma sub-rotina envolve uma série de passos que são dscritos a seguir:
a) Ao executarmos a instrução de chamada de sub-rotina JSR, o endereço de retorno, que é o endereço da instrução imediatamente após a instrução JSR, é salvo automaticamente na pilha, no endereço indicado pelo apontador de pilha (SP);

b) A pilha é uma estrutura de dados em que o último valor a ser colocado é sempre o primeiro a ser retirado e, normalmente, cresce do sentido inverso da memória, isto é, dos endereços mais altos para os mais baixos;
c) Alguns processadores possuem registradores específicos para guardar este endereço, porém o mais comum é que este endereço seja guardado na pilha na memória;
d) A pilha, além de guardar o endereço de retorno, pode ser utilizada para a passagem e o recebimento de parâmetros para a sub-rotina, o que deve ser feito explicitamente pelo programador ou compilador;
e) Depois que o endereço de retorno foi salvo na pilha, o valor do apontador de instruções (PC) é alterado pela instrução JSR para o início da sub-rotina, onde o processador continua a execução das instruções;
f) Se houver parâmetros, os mesmos devem ser retirados da pilha e utilizados pela sub-rotina. Da mesma forma que, ao final da sub-rotina, se houver resultados, eles podem ser passados pelo programador também de volta na pilha;
g) Ao final da sub-rotina, a instrução RET é executada e faz a retirada do endereço de retorno da pilha, no endereço indicado pelo apontador de pilha (SP), copiando o mesmo para o apontador de instruções (PC);
h) A execução do programa prossegue então a partir da instrução seguinte àquela que anteriormente chamou a sub-rotina.

4. Qual a função do ponteiro de pilha (SP - do inglês *stack pointer*)?
   Resposta: O ponteiro de pilha (SP - Stack Pointer) é um registrador especial utilizado em computadores e processadores para gerenciar a pilha de execução do programa. A pilha é uma estrutura de dados na memória utilizada para armazenar informações temporárias, como valores de variáveis locais, endereços de retorno de subrotinas e outros dados relevantes para o fluxo de execução do programa.
   A função do ponteiro de pilha é apontar para o topo da pilha, ou seja, para a posição de memória onde o próximo dado será armazenado. Quando um valor é colocado na pilha (por exemplo, ao chamar uma subrotina ou ao salvar o valor de uma variável local), o ponteiro de pilha é incrementado para apontar para a próxima posição disponível na pilha. Da mesma forma, quando um valor é removido da pilha (por exemplo, ao retornar de uma subrotina ou ao liberar uma variável local), o ponteiro de pilha é decrementado para apontar novamente para o topo da pilha.
   O ponteiro de pilha é fundamental para o correto funcionamento de subrotinas e chamadas de função, pois permite que as informações relevantes sejam armazenadas e recuperadas na ordem correta, garantindo que a execução do programa possa ser retomada corretamente após a conclusão de uma subrotina.
5. Codifique um programa que leia um valor N do painel de chaves, faça a soma dos números entre 1 e N e apresente o resultado no visor hexadecimal. Desconsidere o caso em que acontece *overflow*.
   Resposta: Disponível no repositório do simulador no arquivo soma_1_n.asm
6. Implemente um programa que leia dois valores em sequência do painel de chaves e de acordo com um terceiro valor faça:
   - 0 - Mostre o maior valor no visor hexadecimal.

- 1 - Mostre o menor valor no visor hexadecimal
- 2 - Mostre a soma dos valores no visor hexadecimal.
- 3 - Mostre a diferença dos valores no visor hexadecimal.

Por enquanto, ignore se o resultado tiver *overflow* ou for negativo.

Resposta: Disponível no repositório do simulador no arquivo compara_soma.asm

7. No Exemplo 7.25 apresentamos um programa que calcula a soma dos elementos pares de um vetor. Implemente uma rotina que receba o endereço de um vetor com elementos de 8 bits na pilha e um parâmetro adicional passado no acumulador e de acordo com esse valor faça:
   - 0 - Calcula a soma de todos os elementos pares do vetor
   - 1 - Calcula a soma de todos os elementos ímpares do vetor

   O resultado é devolvido no acumulador.

   Resposta:

```
;---------------------------------------------------
; Rotina que calcula a soma dos elementos pares
; ou ímpares do vetor
; Autores: Antonio Borges e Gabriel P. Silva
; Data:  10/08/2023
; Arquivo: soma_pares_impares.asm
; opção na chave (bit 0)
;---------------------------------------------------
ORG   0
TESTA_SOMA_VETOR:
     LDS   #STACK_ADDR     ; Inicializa a pilha
     LDA   PT_VETOR        ; Carrega o endereço do vetor na
         pilha
     PUSH
     LDA   PT_VETOR+1
     PUSH
     LDA   TAMANHO_VETOR   ; Carrega o tamanho do vetor
     PUSH
ENTRA:
     IN    STATUS          ; Verifica se entrou dado
     AND   #1              ;
     JZ    ENTRA
     IN    CHAVES          ; opção 0: soma elementos pares,
                           ; 1: soma elementos ímpares
     PUSH
     JSR   SOMAVETOR       ; Chama a rotina
     OUT   VISOR           ; Mostra o acumulador no visor
     HLT
;-------------------------------
PT_VETOR:       DW  ORIGEM     ; Ponteiro para a área de origem
ORIGEM:         DB  5,7,2,8,2,1,1,9
TAMANHO_VETOR:  DB  8          ; Tamanho do vetor
STACK:          DS  40         ; Região reservada para a pilha
STACK_ADDR:     DS  0          ; A pilha começa aqui
VISOR           EQU 0
CHAVES          EQU 0
```

```
STATUS          EQU 1
;-----------------------------------------------------------
; Rotina SOMAVETOR
; Na pilha: o endereço de um vetor, o tamanho do vetor e
; a opção
; Opções: 0 soma os pares, 1 soma os ímpares
;-----------------------------------------------------------
SOMAVETOR:
        POP                     ; Salva o endereço de retorno
        STA   RETURN_ADDR       ; Jogado na pilha pelo JSR
        POP
        STA   RETURN_ADDR+1
        POP                     ; Pega a opção de processamento
        STA   OPC
        POP                     ; Pega o tamanho do vetor na pilha
        STA   TAMANHO           ; Salva em tamanho
        POP                     ; Pega o endereço do vetor na pilha
        STA   PVET+1            ; Salva em pvet
        POP
        STA   PVET
        LDA   #0                ; Inicializa a soma
        STA   SOMA
        LDA   #0                ; Inicializa o contador de elementos
        STA   CONTADOR
LOOP:
        LDA   CONTADOR          ; Carrega o contador
        SUB   TAMANHO           ; Compara com o tamanho do vetor
        JZ    FIM_SOMAPAR       ; Se for igual, termina a rotina
        LDA   @PVET             ; Pega o próximo elemento
        AND   #1                ; Testa se é par ou ímpar
        XOR   OPC               ; Realiza a operação "XOR"
        JNZ   SALTA             ; Se não for, pula para SALTA
        LDA   @PVET             ; Faz a soma
        ADD   SOMA
        STA   SOMA
SALTA:
        LDA   PVET              ; Incrementa o ponteiro
        ADD   #1
        STA   PVET
        LDA   PVET+1
        ADC   #0
        STA   PVET+1
        LDA   CONTADOR          ; Incrementa o contador de elementos
        ADD   #1
        STA   CONTADOR
        JMP   LOOP
FIM_SOMAPAR:
        LDA   RETURN_ADDR+1     ; Recoloca na pilha o
        PUSH                    ; endereço de retorno
        LDA   RETURN_ADDR
        PUSH
        LDA   SOMA
```

```
    RET                     ; Sai da rotina
;--------------------------------------------------
; Variáveis da rotina
;--------------------------------------------------
PVET:           DW  0
OPC:            DB  0
SOMA:           DB  0
TAMANHO:        DB  0
CONTADOR:       DB  0
RETURN_ADDR:    DW  0
            END TESTA_SOMA_VETOR
```

8. Implemente um programa que faça a ordenação de um vetor com elementos de 8 bits com sinal.

   Resposta: Disponível no repositório do simulador no arquivo ordena_vetor.asm
9. Implemente uma rotina para somar dois números de 8 bits, devolvendo o resultado no acumulador, e também no visor. Restrições da implementação: esta rotina deve receber os dois números na pilha e devolver a resposta também na pilha.
   Resposta:
   Disponível no repositório do simulador no arquivo soma_8bits.asm
10. Escreva uma rotina que receba o endereço de uma palavra de 16 bits na pilha e que conte o número de '1's na palavra. O resultado é devolvido no acumulador. Apresente um programa de exemplo que mostre o resultado no visor hexadecimal.
    Resposta: Disponível no repositório do simulador no arquivo conta_uns_16.asm
11. Escrever uma rotina com funcionalidade similar ao procedimento *memset*, da biblioteca padrão de C. Ou seja, o procedimento deve preencher uma região de memória com um valor de byte fornecido como parâmetro. Os parâmetros são passados na pilha, ou seja, o endereço inicial, o número de bytes e o valor a ser preenchido. Apresente um programa completo de exemplo.
    Resposta: Disponível no repositório do simulador no arquivo memset.asm
12. Escreva uma rotina para copiar uma cadeia de caracteres de uma posição da memória para outra. Os endereços de origem e destino são passados como parâmetros na pilha e a cadeia de caracteres é terminada com NULL (0x00). Apresente um programa completo de exemplo.
    Resposta: Disponível no repositório do simulador no arquivo copia_cadeia.asm
13. A conversão de binário para decimal é usualmente feita com sucessivas divisões por 10 para extrair os dígitos decimais, da direita para a esquerda. Contudo, o Sapiens não tem a instrução de divisão. Um algoritmo alternativo possível é somar 6 se o número estiver entre 10 e 16. Assim se for 0AH em binário o valor em decimal deve ser 10, e como 0AH + 6 = 10H usa-se este artifício. Se o número estiver entre 20 e 26 soma-se 12, se estiver entre 30 e 36, soma-se 18 e assim por diante. Depois do número corrigido basta então imprimir separadamente os 4 bits mais altos e depois os 4 bits mais baixos, não esquecendo de converter cada um deles para ASCII primeiro. O algoritmo está limitado a valores positivos com um máximo igual a 99 em decimal. Implemente uma rotina que receba um valor em binário no acumulador e que faça a impressão na console e no *banner* do seu valor no formato decimal. Elabore um programa de teste que leia o valor do painel de chaves. Você teria uma sugestão para converter valores até 255?

Resposta: Disponível no repositório do simulador no arquivo converte_binario.asm

14. Escrever um programa que calcule o valor correspondente da série de Fibonacci usando um procedimento recursivo, conforme o exemplo em linguagem de alto nível a seguir.

```c
/*
Função recursiva que recebe um inteiro n >= 1
e devolve o n-ésimo termo da sequência de Fibonacci.
Arquivo: fibonacci.c
*/
#include <stdio.h>
int fibonacci(int n) {
  int x;
  if (n == 1) return(1);
  if (n == 2) return(1);
  x = fibonacci(n-1) + fibonacci(n-2);
  return(x);
}

int main() {  // Função principal.
int n;
  while(n <= 0) {
    printf("Entre com um valor positivo: ");
    scanf("%d", &n);
  }
  printf("Resultado = %d \n", fibonacci(n));
  return(0);
}
```

Resposta: Disponível no repositório do simulador no arquivo fibonacci.asm