

# Parte 1 – Exercícios de Software Básico

## 1.1 Módulo 1 – Compiladores

### Questões Teóricas

1.1.1. Responda sucintamente:

- (a) Qual a diferença mais importante entre uma macro e uma subrotina (função)?
- (b) Durante o processo de tradução são necessários dois estágios, Análise e Síntese. Explique brevemente o objetivo de cada estágio e liste as sub-etapas de cada um deles.

1.1.2. Os itens abaixo podem conter erros. Destaque-os, e corrija-os reescrevendo os itens.

- (a) O bootstrap loader faz parte do Sistema Operacional. Ele é carregado no processo de BOOT. Possui um tamanho sempre múltiplo a 512 bytes e foi substituído pela GPT para ocupar muito espaço em memória RAM.
- (b) As bibliotecas dinâmicas com carregador estático geram um executável chamado de standalone que tem como vantagem ser mais portátil que o executável gerado usando bibliotecas dinâmicas com carregador dinâmico. Porém, o carregador dinâmico é o único que garante que somente tenha uma única cópia da biblioteca em memória.
- (c) O formato de arquivos .COM é um formato sem cabeçalho de no máximo 64 kB que não permite ligação nem debug. É um arquivo tanto para objeto como executável. O formato COFF também é formato de objeto executável, porém ele permite bibliotecas dinâmicas e debug.
- (d) A principal vantagem de uma linguagem compilada em relação a uma interpretada é a otimização completa do código sem precisar manter relação de linha de código compilado com o arquivo de texto de entrada com o programa original. Porém, uma outra vantagem é que os programas compilados são mais portáteis, já que o arquivo executável de um programa compilado pode ser executado em qualquer Sistema Operacional.

1.1.3. Abaixo estão listadas várias afirmativas incorretas. Justificando, identifique os erros, e corrija-os.

- (a) O formato .COM caracteriza-se por ter um endereço fixo (100H) para o ponto de entrada do programa. O cabeçalho de um arquivo nesse formato possui tamanho reduzido simplesmente composto pelos caracteres 'MZ' e a quantidade de segmentos de 64 kiB necessários para esse programa.
- (b) Bootstrap loader é um carregador especial, já que ele consiste em um programa armazenado completamente em um único setor conhecido como MBR.
- (c) A utilização de bibliotecas dinâmicas permite que um trecho de código chamado por vários problemas possa ter uma única cópia em memória, e somente carregada ao ser executada: o montador e o ligador portanto não necessitam serem informados sobre o uso de uma biblioteca dinâmica.
- (d) O formato ELF é um formato de exclusivo de arquivos objeto complexo que armazena tabelas com informações de realocação para o ligador. O formato PE (portable executable) é um formato feito em base no formato ELF.

1.1.4. Abaixo estão listadas várias afirmativas, as quais podem estar erradas. Identifique o(s) erro(s), dê o por quê(s), e corrija-o(s).

- (a) Executável ligado com biblioteca estática possui a vantagem de ser mais portátil que o executável com biblioteca dinâmica. O programa também é carregado mais rápido em memória. Porém, é possível ter várias cópias da mesma biblioteca em memória. Por esse motivo os formatos de arquivos mais recentes como ELF e PE não permitem ligação estática.
- (b) A utilização de bibliotecas dinâmicas com carregador estático permite que um trecho de código chamado por vários programas possa ter uma única cópia em memória, e somente carregada ao ser executada. O montador e o ligador portanto não necessitam serem informados sobre o uso de uma biblioteca dinâmica.
- 1.1.5. Descreva as informações contidas no MBR, indicando suas partes. Indique qual é o objetivo do setor MBR no processo de carregação do Sistema Operacional. Indique onde se encontra o MBR. Descreva por que o MBR foi substituído pelo GPT em sistemas computacionais modernos.
- 1.1.6. Explique para que serve o código de 3 endereços durante o processo de compilação; se é usado durante a fase de análise ou síntese e; qual a diferença entre a tabela de 3 colunas e a de 4.
- 1.1.7. Dado o código de três endereços abaixo, que trabalha com array de bytes em memória, responda:

```

(01) i = 0                                     1
(02) if i >= n goto (14)                       2
(03) j = 0                                     3
(04) if j >= n goto (12)                       4
(05) t1 = n * i                               5
(06) t2 = t1 + j                               6
(07) t3 = t2 * 8                               7
(08) c1[t3] = 0.0                             8
(09) m = n * n                                 9
(10) j = j + 1                                10
(11) goto (4)                                 11
(12) i = i + 1                                12
(13) goto (2)                                 13
(14) i = 0                                    14
(15) if i >= m goto (19)                      15
(16) c2[i] = 0                                16
(17) i = i + 1                                17
(18) goto (15)                                18

```

- (a) O código foi otimizado? Justifique.
- (b) Assumindo que o código inicial era ANSI C, qual o tipo da variável c2 e qual o tipo da variável c1? Justifique.
- 1.1.8. Cada um dos seguintes trechos de código em linguagem C possui um ou mais erros. Indique onde estão os erros e classifique-os como léxico, sintático ou semântico.

- (a) `for (a=2, b=50; a<40; a++, b>50)` 1  
`printf("%d, %d", a, b);` 2
- (b) `double x=25.1, y=10.0, *z;` 1  
`z = &x;` 2  
`if (x == z) x += y;` 3

```
(c) switch (a) {
    case 1: f1();
        continue;
    case 2: f2();
}
```

```
(d) float a, b, c=0;
    int 3d = 4;
    scanf("%d", &a);
    b = a % c;
```

- 1.1.9. As funções em C abaixo fazem parte do mesmo programa, como um arquivo de cabeçalho (\*.h). O arquivo não compila. Indique as linhas erradas, explicitando se o erro é léxico, sintático ou semântico. Assuma que CHAR\_BIT foi definido corretamente.

```
int abs(int *a) {
    int b = a;
    b = (b >> (sizeof(int)*CHAR_BIT - 1) & 1);
    return 2 * b * (a) + a;
}

int max(int a, int b) {return (a + b + abs(a - b)) / 2;}
int max(int a, int b) {return (a + b - abs(a - b)) / 2;}

void _sort(int &a, int &b, int &c) {
    int maxnum = max(max(a, b), c);
    int minnum = min(min(a, b), c);
    int middlenum = a + b + c - maxnum - minnum;

    if (a == b)
        if (a == c)
            printf("all numbers are equal");
    a = maxnum;
    b = middlenum;
    c = minnum;
}
```

- 1.1.10. Detecte os erros no código abaixo, sublinhando o erro e indicando se o erro é sintático, léxico ou semântico.

```
#include <stdio.h>

int n = 100; // global
void print_plus_n(int x) {printf(" %d ", x + n);}
void increment_n() {n = n + 1;}
int main() {
    int n, i;
    int *a, b##2;
    n = 1;
    print_plus_n(25);
    n = 33;
```

```

print_plus_n(n);
increment_n();
printf("%d", n);
print_plus_n(n);
if (print_plus_n > 100) {
    *(a) = n;
    increment_n();
    break;
}
for (i=0; i++)
    increment_n();
print_plus_n(n);
}

```

1.1.11. No código abaixo identifique os erros e indique se é erro léxico, semântico ou sintático.

```

#include <stdio.h>
int main() {
    int m, n, p, q, c, d, k, s, m, sum=0;
    int first[10][10], second[10][10], multiply[10][10];
    char t, m=0;

    printf("Enter the number of rows and columns of first matrix\n");
    scanf("%d %d", &m, &n);
    printf("Enter elements of first matrix\n");

    for (c=0; c<m; c++)
        for (d=0; d<n; d++)
            scanf("%d", &first[c][d]);

    printf("Enter number of rows and columns of second matrix\n");
    scanf("%d %d", &p, &q);

    if (*n != p) {
        printf("The multiplication isn't possible.\n");
        break;
    }
    else {
        printf("Enter elements of second matrix\n");
        for (c=0; c<p; c++)
            for (d=0; d<q; d++)
                scanf("%d", &second[c][d]);

        for (c=0; c<m; c++) {
            for (d=0; d<q; d++) {
                for (k=0; k<p) {
                    sum = sum + first[c][k] * second[k][d];
                }
                multiply[c][d] = sum;
                sum = 0;
            }
        }
    }
}

```

```

    }
    printf("Product of the matrices:\n");

    for (c=0; c<m; c++) {
        for (d=0; d<q; d++)
            printf("%d\t", multiply[c][d]);
        printf("\n");
    }
}
return 0;
}

```

1.1.12. Indique quais são os erros no código abaixo, classificando-o como léxico, sintático ou semântico.

```

#include <stdio.h>
int main() {
    int array[100], c, d, swap;
    int D&;

    printf("Enter number of elements\n");
    scanf("%d", &n);
    printf("Enter %d integers\n", n);

    for (c=0; c<n; c++)
        scanf("%d", &array[c]);

    for (c=0; c<n-1; c++) {
        for (d=0; d<n-c-1; d++) {
            if (array[d] > array[d+1]) {
                swap = array[d];
                array[d] = array[d+1];
                array[d+1] = swap;
            }
        }
    }
    printf("Sorted list in ascending order:\n");
    for (c=0; c<n)
        printf("%d\n", array[c]);
    return 0;
}

```

## Questões Práticas

1.1.13. Dado o programa abaixo,

```

SECTION TEXT
M1: MACRO    &A, &B, &C
COPY        &A, &B

```

```

    INPUT    &C                                4
    OUTPTUT  &B                                5
ENDMACRO                                        6
M2: MACRO   &A, &B, &C                        7
    OUTPUT  &A                                8
    COPY    &B, &C                            9
    COPY    &A, &B                           10
ENDMACRO                                        11
COPY       ZERO, OLDER                        12
M1         ONE, OLD, LIMIT                    13
FRONT:     LOAD    OLDER                      14
           ADD     OLD                          15
           STORE   NEW                          16
           SUB     LIMIT                       17
           JMPP    FINAL                       18
           M2      NEW, OLD, OLDER             19
           JMP     FRONT                       20
FINAL:     OUTPUT  LIMIT                      21
STOP                                              22
SECTION DATA                                    23
ZERO:     CONST  0                             24
ONE:      CONST  1                             25
OLDER:    SPACE                                    26
OLD:      SPACE                                    27
LIMIT:    SPACE                                    28
NEW:      SPACE                                    29

```

- (a) Mostre como ficaria a MNT e a MDT, explicando o que são essas tabelas e para que são usadas..
- (b) Mostre a tabela de símbolos assumindo que foi utilizado o algoritmo de passagem única, escrevendo as listas de pendências. (note que é necessário primeiro resolver as macros).

1.1.14. Dado o código abaixo em assembly inventado visto em sala de aula, mostre o arquivo objeto desse programa, chamado PROG1. Coloque T na frente de cada linha da parte de texto, e H na frente de cada linha de todos os headers (caso necessário). Em cada linha de header, indique textualmente o significado do conteúdo. Caso necessário, informação de realocação pode ser dada utilizando os 2 formatos vistos em aula para arquivos reais.

```

SECTION TEXT                                    1
M_2:     MACRO                                    2
           JMP     FAT                            3
FIM:     OUTPUT  N                                4
ENDMACRO                                        5
MUL_N:   MACRO                                    6
           MUL     N                                7
           STORE   N                                8
           LOAD    AUX                              9
           M_2                                         10
END_MACRO 11
INPUT    N                                         12
LOAD     N                                         13
FAT:     SUB      ONE                              14

```

```

        JMPZ    FIM                15
        STORE   AUX                16
        MUL_N                      17
STOP                                           18
SECTION DATA                               19
AUX:     SPACE                        20
N:       SPACE                        21
ONE:     CONST 1                      22

```

1.1.15. Considere os módulos a seguir na Linguagem de Montagem Hipotética apresentada em sala.

MOD_A: BEGIN	1		
Y: EXTERN	2	MOD_B: BEGIN	1
MOD_B: EXTERN	3	VAL: EXTERN	2
PUBLIC VAL	4	L1: EXTERN	3
PUBLIC L1	5	PUBLIC Y	4
INPUT Y	6	PUBLIC MOD_B	5
LOAD VAL	7	OUTPUT Y	6
ADD Y	8	OUTPUT VAL	7
STORE Y + 2	9	OUTPUT Y + 2	8
JMPP MOD_B	10	JMP L1	9
L1: STOP	11	Y: SPACE 3	10
VAL: CONST 5	12	END	11
END	13		

- Personifique um montador e monte os módulos como uma sequência de números inteiros, utilizando a tabela em apêndice como referência. Apresente o código montado e as tabelas resultantes.
- Personifique um ligador e combine os módulos em um único arquivo executável. Apresente o código ligado indicando os endereços absolutos e relativos, escrevendo no cabeçalho antes do código um mapa de bits mediante a diretiva R (ex.: R 0101010001).

1.1.16. Considere os módulos a seguir na Linguagem de Montagem Hipotética apresentada em sala.

MOD_A: BEGIN	1		MOD_C: BEGIN	1
Y: EXTERN	2		_L2: EXTERN	2
MOD_B: EXTERN	3	MOD_B: BEGIN	Y: EXTERN	3
PUBLIC VAL	4	MOD_C: EXTERN	VAL: EXTERN	4
PUBLIC _L2	5	ONE: EXTERN	PUBLIC MOD_C	5
PUBLIC ONE	6	PUBLIC Y	OUTPUT Y	6
INPUT Y	7	PUBLIC MOD_B	OUTPUT VAL	7
_L1: JMP MOD_B	8	LOAD Y	JMP _L2	8
_L2: LOAD VAL	9	ADD ONE	END	9
SUB ONE	10	STORE Y		
STORE VAL	11	JMP MOD_C		
JMPP _L1	12	Y: SPACE		
STOP	13	END		
VAL: CONST 5	14			
ONE: CONST 5	15			
END	16			

- (a) Personifique um montador e monte os módulos. Apresente as tabelas de símbolos, de uso e de definição de cada módulo (não é necessário apresentar o código montado de cada módulo).
- (b) Personifique um ligador e combine os módulos em um único arquivo executável. Somente é necessário apresentar o código ligado final indicando no código os endereços absolutos e relativos e os fatores de correção. O módulo A deve ir primeiro no código final, seguido de B e finalmente o C.

1.1.17. Utilizando a linguagem Assembly hipotética vista em sala de aula:

- (a) Faça um programa que receba um número inteiro (de 16 bits) do usuário e escreva na tela uma sequência de 1s e/ou 0s sendo a representação binária do número indicado pelo usuário (do bit menos significativo ao mais significativo).
- (b) Personifique um montador de passagem única e realize a montagem do programa. Apresente a Tabela de Símbolos resultante e o código máquina, ambos antes de resolver as referências pendentes.

1.1.18. Utilizando a linguagem Assembly hipotética vista em sala de aula:

- (a) Faça um programa em assembly que receba 1 número (positivo) do usuário, e verifique se é múltiplo de 3: se sim mostrar 1 na tela, caso contrário mostrar 0. A parte de dados deve ir depois da parte de código. A verificação se o número é múltiplo ou não deve ser feito utilizando uma macro que recebe como argumento o endereço de memória onde foi salvo o número digitado pelo usuário. O que tem de ser feito fora da macro é a leitura do número e a impressão na tela da saída.
- (b) Personifique um montador de passagem única e realize a montagem do programa. Apresente a Tabela de Símbolos resultante e o código máquina antes de resolver as referências pendentes (ou seja mostrar a tabela de símbolos e as listas de pendências). Não precisa apresentar o código montado nesta parte.

1.1.19. É preciso fazer um programa em Assembly formado por 2 módulos. O programa deve operar da seguinte forma:

1. O primeiro módulo deve pedir ao usuário 2 números. Deve armazenar esses dois dígitos em memória (rótulo) que foi reservada para receber os 2 valores. Ou seja, um só rótulo foi reservado para 2 endereços de memória. Esse rótulo deve ter sido reservado no módulo 2.
  2. Após receber e salvar os 2 números, o primeiro módulo deve pular para o segundo módulo.
  3. O segundo módulo deve verificar se o segundo número é diferente de zero. Se for diferente de zero, deve voltar ao módulo 1.
  4. O módulo 1 deve então mostrar a divisão do primeiro pelo segundo.
- (a) Mostre os 2 módulos utilizando o Assembly inventado visto em sala de aula. Cada módulo sempre deve ter a seção de dados depois da seção de texto. Não é necessário utilizar a diretiva SECTION para dividir as seções, mas é obrigatório o uso de BEGIN e END.
  - (b) Mostre a tabela de símbolos, uso e definições de cada módulo, assim como o fator de correção de cada módulo.
  - (c) Mostre o código objeto após os módulos terem sido ligados.

## 1.2 Módulo 2 – Assembly x86-64

### Questões Teóricas

1.2.1. Dadas as seguintes instruções e os seus respectivos códigos de máquina, indique os valores dos campos OPCODE, Mod R/M, SIB, DISPLACEMENT, e IMMEDIATE. Note que uma instrução pode deixar de apresentar algum campo.

- (a) `mov edx, 0x0` | `ba 00 00 00 00`



(b) `mov ebp, esp | 89 e5`

1.2.2. Sobre a arquitetura x64, responda:

- (a) O que significa um processador ser de arquitetura híbrida RISC/CISC?
- (b) Descreva como é feito o endereçamento de memória na arquitetura x64, indicando os grupos de bits dentro do endereçamento virtual.
- (c) Explique brevemente o que é a tecnologia SIMD, indicando os registradores envolvidos.
- (d) Indique as diferenças entre os registradores de uso geral da arquitetura IA-32 e x64.

1.2.3. Descreva as diferenças sobre endereçamento e utilização de memória do Modo Real e o Modo Protegido. Para o modo real, indique somente como é calculado o endereço de memória. No modo protegido, faça um diagrama mostrando os diferentes segmentos de memória, indique como é calculado o endereço lógico e real, e por que o modo é chamado de protegido.

1.2.4. Uma instrução de pulo (ou salto) pode ser classificada de diversas formas. Responda sucintamente às perguntas abaixo com relação a pulos.

- (a) O que significa um pulo curto relativo? Como é calculado o valor no contador de programa após executar a instrução de pulo?
- (b) O que significa um pulo distante absoluto indireto? Como é calculado o valor no contador de programa após executar a instrução de pulo?
- (c) Quais os tipos de pulos distantes no Protected Mode e Real Mode?

1.2.5. Os itens abaixo possuem instruções de programas Assembly IA-32 (em modo nativo) que utilizam diversos modos de endereçamento. Classifique cada item como correto ou errado, e justifique o que estiver errado.

- (a) `mov eax, 10`
- (b) `mov [M], al`
- (c) `mov al, [cs + esi + array]`
- (d) `mov vetor[1], 0`
- (e) `add ax, [X + ecx]`
- (f) `mov esi, vetor + ebx`
- (g) `inc WORD [inicio + ebx*8 + esi]`
- (h) `mov [ebx + esi*4], DWORD 5`
- (i) `dec BYTE [bl]`
- (j) `add [x + 1], al`
- (k) `mov eax, [array + ecx*8 + ebx]`
- (l) `mov [eax*8 + 1], 5`
- (m) `mov bl, ax`
- (n) `cmp [esi], 10`
- (o) `adc al, ah`

1.2.6. Existem três tipos básicos de operandos: imediato, registrador e memória. O acesso a memória pode ser feito de duas maneiras: direta ou indireta. Em cada instrução com algum tipo de endereçamento do código abaixo especifique que tipo de operando está sendo usado como fonte e destino: imediato, memória direta/indireta, ou registrador. Indicar se algum endereçamento é ilegal.

<code>section .data</code>	1
<code>count db 2</code>	2
<code>wList dw 0003h, 2000h</code>	3
<code>array db 0Ah, 0Bh, 0Ch, 0Dh</code>	4
	5
<code>section .text</code>	6
<code>global _start</code>	7

```

_start:
    mov esi, wList
    mov ax, [esi]
    mov bx, ax
    mov al, [array + bx]
    add al, [count]
    mov ax, 40
    sub ax, [wList + 2]
    mov [count], 50
    mov eax, 1
    mov ebx, 0
    int 80h

```

1.2.7. Considere o seguinte fragmento de código IA-32 obtido pelo comando `objdump`. Como pode ser visto, a função `g()` é chamada pela função `f()`.

```

08048384 <g>:
    8048384    55                push ebp
    8048385    89 e5                mov ebp, esp
    8048387    8b 45 0c             mov eax, [ebp + 0xc]
    804838a    03 45 08             add eax, [ebp + 0x8]
    804838d    5d                pop ebp
    804838e    c3                ret

0804838f <f>:
    804838f    55                push ebp
    8048390    89 e5                mov ebp, esp
    8048392    83 ec 08            sub esp, 0x8
    8048395    c7 44 24 04 18 00 00 mov [esp + 0x4], 0x18
    804839c    00
    804839d    c7 04 24 0c 00 00 00 mov [esp], 0xc
    80483a4    e8 db ff ff ff      call 8048384 <g>
    80483a9    89 ec                mov esp, ebp
    80483ab    5d                pop ebp
    80483ac    c3                ret

```

- Quantos argumentos cada uma das funções `f()` e `g()` recebem?
- Quando o CPU está a ponto de executar a instrução `add` no endereço `0x0804838a` em `g()`, mostre os valores na pilha, preenchendo a tabela abaixo.

esp + 12	
esp + 8	
esp + 4	
esp	

## Questões Práticas

- 1.2.8. O programa abaixo realiza a cópia de um vetor de *double words*, convertendo-o de *little endian* para *big endian*. Complete o programa, indicando as instruções dos espaços em branco (cada espaço deve ser preenchido com uma única instrução).

```

SIZE EQU 6
section .data
little dd 42434445h, 45454545h, 4A4B4C4Dh,
        dd 414D4E4Fh, 46454948h, 4C474D46h

section .bss
big resd SIZE
temp resd 1

section .start
global _start
_start:
    mov ecx, SIZE
    mov eax, little
    mov esi, big
laco1: mov ebx, esi
    -----
laco2: mov dl, [eax]
    mov [ebx], dl
    dec ebx
    inc eax
    -----
    jae laco2
    add esi, 4
    -----
    cmp ecx, 0
    -----
done: mov eax, 1
    mov ebx, 0
    int 80h

```

- 1.2.9. Para cada código C abaixo, escreva o equivalente em Assembly IA-32. Diretivas em C **devem** ser substituídas por diretivas equivalentes em IA-32. Use os registradores para as variáveis locais (com exceção de estruturas de dados) e seção de Dados ou BSS para as variáveis estáticas ou globais. **Deve-se** utilizar os endereçamentos corretos para cada tipo de estrutura de dados. Não se preocupe pelo fato do programa principal em C ser uma função.

(a)

```

#define MAX 100
int main() {
    int a[100], i;
    for (i=0; i<MAX; i++) a[i] = i>>1;
    return 0;
}

```

```

(b) #include <stdio.h>
#define ROW 5
#define COL 5
int main() {
    int array1[ROW][COL] = {
        {1, 89, 99, 91, 92},
        {79, 2, 70, 60, 55},
        {70, 60, 3, 90, 89},
        {60, 55, 68, 4, 66},
        {51, 59, 57, 2, 5}
    };
    int array2[ROW][COL] = {
        {1, 2, 3, 4, 5},
        {1, 2, 3, 4, 5},
        {1, 2, 3, 4, 5},
        {1, 2, 3, 4, 5},
        {1, 2, 3, 4, 5}
    };
    int array3[ROW][COL];
    int i, j;
    for (i=0; i<ROW; i++)
    for (j=0; j<COL; j++) {
        if (array1[i][j] == array2[i][j]) array3[i][j] = '1';
        else array3[i][j] = '0';
        printf("%c", array3[i][j]);
    }
    return 0;
}

```

```

(c) #define SIZE 11
int main() {
    int vetor[SIZE] = {
        0x10002231, 0x80154491, 0x91929394,
        0x11223344, 0x12131415, 0x79270601,
        0x55127380, 0x16112212, 0x39089607,
        0x51557721, 0x16846676
    };
    int res=0;
    int i=0;
    while (i < SIZE)
        res += vetor[i++];
    return 0;
}

```

(d) Não é permitido MUL ou IMUL.

```

#define MAX 100
int main() {
    short int a[MAX][MAX];
    int i, j;
    for (i=0; i<MAX; i++)

```

```

    for (j=0; j<MAX; j++) {
        if (i == j) a[i][j] = 3*i;
        else a[i][j] = 7*i;
    }
    return 0;
}

```

- (e) Assuma que o usuário vai digitar um número de 0 a 9.

```

char start;
int count;

int main() {
    char sum=0;
    count = 100;
    scanf("%d", &start);
    while (count) {
        if (sum%2) sum += start;
        else sum -= start;
        start++;
        count--;
    }
    return sum;
}

```

1.2.10. Escreva uma versão em C de cada uma das funções em Assembly IA-32 (todas as funções em Assembly colocam o valor de retorno em EAX) seguindo as seguintes regras:

- (i) todas as funções são compostas por uma **única instrução RETURN** sem criar variáveis locais,
- (ii) em nenhuma função em C pode ser utilizado deslocamento de bits ( $\ll$  ou  $\gg$ ) e
- (iii) somente pode ser feita **uma única operação aritmética e/ou uma única comparação** ( $>$  ou  $<$ ) nas funções em C.

(a)

```

foo1:
    push ebp
    mov ebp, esp
    mov edx, [ebp + 8]
    mov eax, edx
    shl eax, 3
    sub eax, edx
    pop ebp
    ret

```

(b)

```

foo2:
    enter 0, 0
    mov eax, [ebp + 8]
    shr eax, 31
    leave
    ret

```

```
(c) foo3:
push ebp
mov ebp, esp
mov eax, [ebp + 8]
mov eax, [eax]
add eax, eax
pop ebp
ret
```

```
(d) foo4:
push ebp
mov ebp, esp
mov ax, [ebp + 8]
sub ax, [ebp + 10]
pop ebp
ret
```

- 1.2.11. O código abaixo em C chama uma função em Assembly IA-32 que retorna o valor da multiplicação entre todos os elementos de um vetor. Escreva essa função em Assembly utilizando laços.

```
#include <stdio.h>
int main() {
    int resa, resb;
    short int a=[1,2,3,4,5,6,7,8,9,10], b=[-1,10,-3,8,-5,6,-7,4,-9,2];
    extern int f4(short *x, int n);
    resa = f4(a, 10);
    resb = f4(b, 10);
    printf("O resultado de A: %d", resa);
    printf("O resultado de B: %d", resb);
    return 0;
}
```

- 1.2.12. O código abaixo em C, chama uma função em Assembly IA-32 que retorna o produto interno entre uma matrix 1xN e outra Nx1. Ela recebe o ponteiro dos dois arrays unidimensionais e o tamanho N deles. Assuma que o primeiro array é 1xN e o segundo é Nx1. Escreva essa função em Assembly (utilize laços).

```
#include <stdio.h>
int main() {
    int resa, resb;
    short int a=[1,2,3,4,5,6,7,8,9,10], b=[-1,10,-3,8,-5,6,-7,4,-9,2];
    extern int f4(short *x, short *y, int n);
    resa = f4(a, b, 10);
    printf("O resultado de A: %d", resa);
    return 0;
}
```

- 1.2.13. O programa em C abaixo solicita ao usuário os valores de uma matriz 10x10 e depois calcula a soma dos elementos da diagonal principal. Altere o programa para uma versão em que o programa

principal continua em C mas a função `soma()` esteja em Assembly IA-32. A função deve calcular a soma dos elementos da diagonal principal, recebendo o ponteiro da matriz, o tamanho dela (NxN), e a variável de retorno da soma como parâmetros mediante a pilha. Mostre o código Assembly da função e indique se é necessário fazer alguma alteração no código principal em C.

```
#include <stdio.h> 1
#define N 10 2
3
void soma(int M[][N], int *valor) { 4
    *valor=0; 5
    int i; 6
    for (i=0; i<N; i++) 7
        *valor += M[i][i]; 8
} 9
10
int main() { 11
    int A[N][N]; 12
    int i, j, res; 13
    for (i=0; i<N; i++) 14
        for (j=0; j<N; j++) { 15
            printf("Digite elemento A[%d][%d]: ", i+1, j+1); 16
            scanf("%d", &A[i][j]); 17
        } 18
    soma(A, &res); 19
    printf("Soma dos elementos da diagonal principal: %d\n", res); 20
    return 0; 21
} 22
```

- 1.2.14. Escreva o seguinte código em Assembly IA-32. Para isso é permitido o uso da biblioteca `io.mac` para ler e escrever strings (`PutStr`, `GetStr`). O programa em Assembly deve mostrar todas as mensagens indicadas no programa em C. Deve manter os ponteiros para arquivo em memória. Assuma que ambos arquivos, entrada e saída, já existem (o arquivo de saída existe, mas está vazio).

```
#include <stdio.h> 1
#define BUF_SIZE 256 2
3
int main() { 4
    FILE *fd1, *fd2; 5
    char file_in[30]; 6
    char file_out[30]; 7
    char buf[BUF_SIZE]; 8
9
    printf("Digite o nome do arquivo de entrada: "); 10
    scanf("%s", file_in); 11
12
    printf("Digite o nome do arquivo de saída: "); 13
    scanf("%s", file_out); 14
15
    fd1 = fopen(file_in, "r"); 16
    fd2 = fopen(file_out, "w"); 17
```

```

fread(buf, sizeof(char), BUF_SIZE, fd1);
fwrite(buf, sizeof(char), BUF_SIZE, fd2);

fclose(fd1);
fclose(fd2);

return 0;
}

```

1.2.15. Faça um programa em Assembly IA-32 que:

1. Abre um arquivo em modo leitura (00), com permissão para todos os usuários possam ler, escrever e executar. Nome do arquivo: 'myfile1.txt' (na mesma pasta que o programa). Lê  $n$  valores de char do arquivo e os salva num array  $x$ . Fecha o arquivo.
2. Soma todos os elementos do array.
3. Abre um arquivo em modo escrita (01), com permissão para somente o dono do arquivo ler, escrever e executar. Nome do arquivo: 'myfile2.txt' (na mesma pasta que o programa). Escreve no arquivo o valor da soma. Fecha o arquivo.

1.2.16. Faça um programa em Assembly IA-32 que multiplique duas matrizes 5x5 de inteiros. O programa deve primeiro fazer um laço para preencher a primeira matriz a ser digitada pelo usuário. Depois deve fazer um laço para preencher a segunda matriz digitada pelo usuário. Em seguida, deve multiplicar as duas matrizes e salvar o resultado em outra matriz. Utilize o endereçamento correto para matrizes. Os labals das matrizes devem ser declarados no SECTION BSS. Não é necessário imprimir mensagens para o usuário, nem mostrar o resultado final.

1.2.17. Considere um número em ponto flutuante baseado no formato da IEEE. O número é formado por 6 bits. Um bit para o sinal, os próximos três bits para o expoente, e os últimos dois bits para a mantissa. Como visto em sala de aula, o formato IEEE possui números normalizados, não normalizados, duas representações de zero, infinito e NaN. Assumindo que arredondamentos são feitos utilizando o arredondamento ao inteiro mais infinito (*ceil*), preencha a tabela abaixo nos campos *binário*, *mantissa*, *expoente* e *valor*. No campo *binário*, deve-se colocar o binário do número completo, enquanto que nos outros campos deve-se colocar números decimais. Pode utilizar notação exponencial ( $2^{512}$ ) ou fracionária ( $2\frac{1}{3}$ ). Quando pede-se o maior/menor número, não devem ser considerados os infinitos.

Descrição	Binário	Mantissa	Expoente	Valor decimal
Menos zero	100000	0	-2.0	-0.0
Número positivo mais próximo a zero				
Infinito negativo				
Maior número normalizado				
Menor número não-normalizado				
$5.0 - 0.75$				
$4.0 + 3.0$				

1.2.18. Considere um número em ponto flutuante baseado no formato da IEEE. O número é formado por 7 bits. Um bit para o sinal, os próximos quatro bits para o expoente, e os últimos dois bits para a mantissa. Como visto em sala de aula, o formato IEEE possui números normalizados, não normalizados, duas representações de zero, infinito e NaN. Assumindo que arredondamentos são feitos utilizando o arredondamento à fração par mais próxima (*round*), preencha a tabela abaixo nos campos *binário*, *mantissa*, *expoente* e *valor*. No campo *binário*, deve-se colocar o binário do número completo, enquanto que nos outros campos deve-se colocar números decimais. Pode



utilizar notação exponencial ( $2^{512}$ ) ou fracionária ( $2\frac{1}{3}$ ). Quando pede-se o maior/menor número, não devem ser considerados os infinitos.

Descrição	Binário	Mantissa	Expoente	Valor decimal
Menos zero	1000000	0	-2.0	-0.0
Número positivo mais próximo a zero				
Maior número normalizado				
Menor número não-normalizado				
$4.0 + 3.0$				

- 1.2.19. Considere um número em ponto flutuante de 9 bits baseado na representação IEEE (segue as regras para números normalizados, não normalizados, representação de 0, infinito e NaN), sendo que existe 4 bits para o expoente e 4 bits para mantissa. Preencha a tabela abaixo. Se for necessário, arredonde para a fração mais próxima. No campo valor pode usar números fracionários (por exemplo,  $\frac{3}{4}$ ) ou inteiros por potência de 2 (por exemplo  $3 \times 2^{-3}$ ).

Número	Valor	Bit sinal	Bits expoente	Bits mantissa
Zero	0.0	0	0000	0000
Negativo mais próximo a zero				
Maior positivo				
n/a	-5.0			
n/a	$1\frac{9}{16} \times 2^{-2}$			
Menos um	-1.0			
O resultado de $4 - 1\frac{9}{16}$				

- 1.2.20. Considere um número em ponto flutuante baseado no formato da IEEE. O número é formado por 10 bits. Um bit para o sinal, os próximos cinco bits para o expoente, e os últimos quatro bits para a mantissa. Como visto em sala de aula, o formato IEEE possui números normalizados, não normalizados, duas representações de zero, infinito e NaN. Assumindo que arredondamentos são feitos utilizando o arredondamento para o par mais próximo, preencha a tabela abaixo nos campos *binário*, *mantissa*, *expoente* e *valor*. No campo *binário*, deve-se colocar o binário do número completo, enquanto que nos outros campos deve-se colocar números decimais. Pode utilizar notação exponencial ( $2^{512}$ ) ou fracionária ( $2\frac{1}{3}$ ). Quando pede-se o maior/menor número, não devem ser considerados os infinitos. A base do expoente é 2.

Descrição	Binário	Mantissa	Expoente	Valor decimal
Número negativo mais próximo a zero				
Maior número				
Menor número não-normalizado				
Menos um				