**UNIVERSIDADE DE SÃO PAULO**

**ESCOLA DE ARTES, CIÊNCIAS E HUMANIDADES**

**BACHARELADO EM SISTEMAS DE INFORMAÇÃO**

**ORGANIZAÇÃO E ARQUITETURA DE COMPUTADORES I**

JÚLIA DU BOIS ARAÚJO SILVA

MARCOS MEDEIROS DA SILVA FILHO

**RELATÓRIO DE ORGANIZAÇÃO E ARQUITETURA DE COMPUTADORES I**

SÃO PAULO - SP

2024

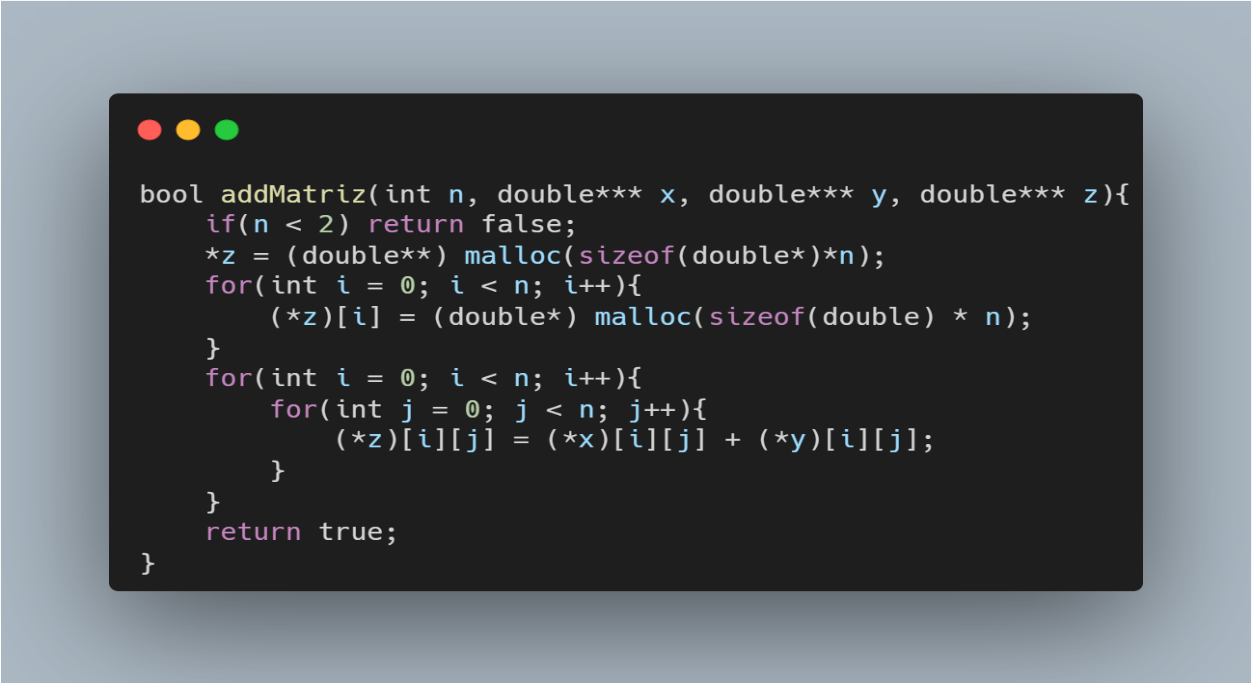
1. **ORGANIZAÇÃO E ARQUITURA MIPS**
2. **DESCRIÇÃO DO PROBLEMA E CÓDIGO ALTO NÍVEL**

Para entregar o trabalho e obter uma maior compreensão sobre a linguagem de montagem MIPS, foram desenvolvidas as soluções dos itens A, B e C do problema 27:

1. Faça uma função que recebe como parâmetros um inteiro e duas matrizes quadradas reais e de ordem . Esta função devolve em uma matriz , também passada como parâmetro, a soma das matrizes e ;
2. Escreva uma função que recebe como parâmetro um número inteiro , um número real e uma matriz . A função devolve em uma matriz , também passada como parâmetro, o produto do número pela matriz . Ou seja, ;
3. Escreva uma função que recebe como parâmetros um inteiro e duas matrizes quadradas reais e . Esta função devolve em uma matriz , também passada como parâmetro, o produto das matrizes e .

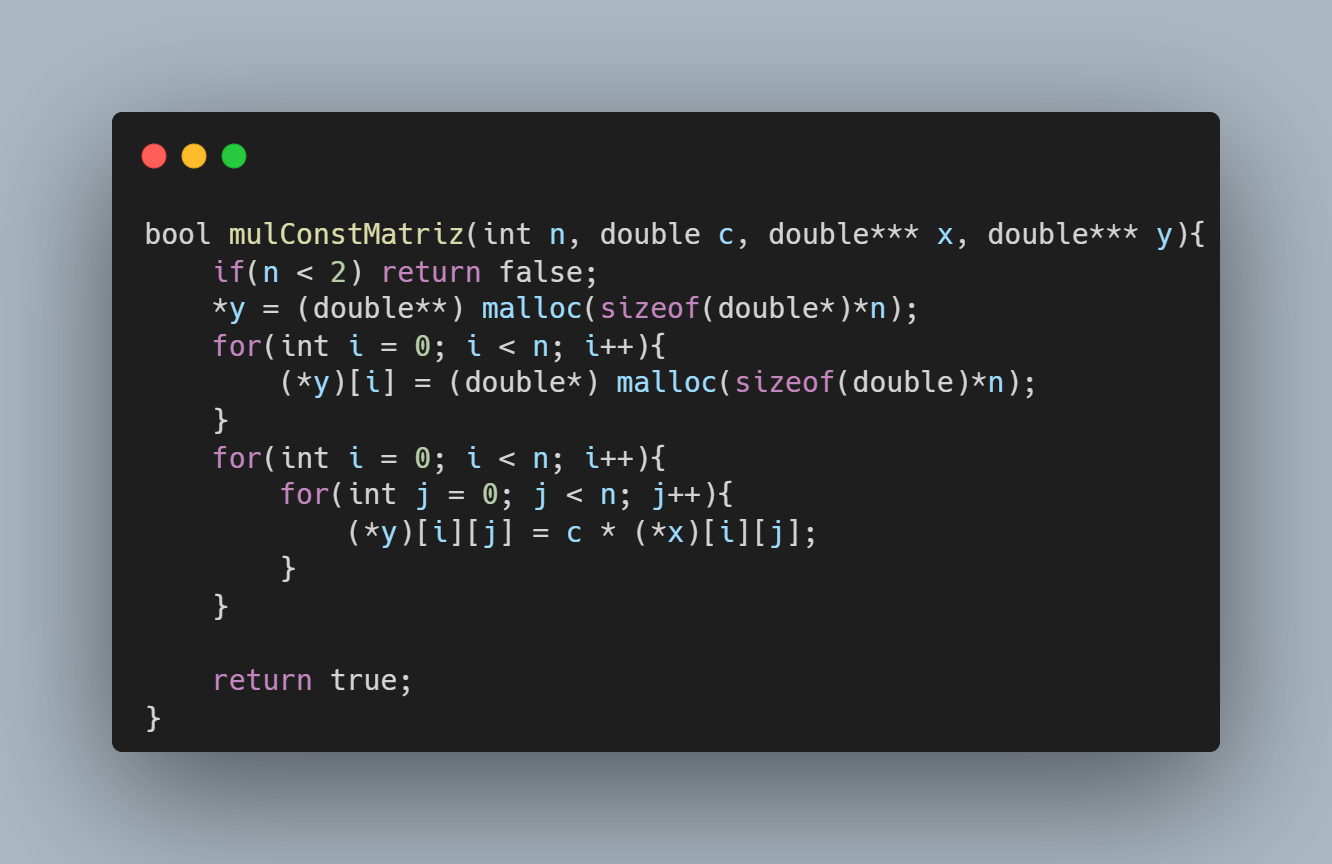
Nesse sentido, o item A solicita que se desenvolva uma função, o qual recebe como parâmetros um inteiro , que representa a dimensão da matriz quadrada real , dois ponteiros para as matrizes reais e e um ponteiro que receberá uma matriz quadrada real resultante da operação da função. Sob essa ótica, a função realizará a operação de soma entre as matrizes e e atribuirá a matriz resultante no ponteiro passado como parâmetro, ou seja, .

Com isso, foi desenvolvido tal função utilizando a linguagem C, o qual se chama “addMatriz”, em que se realiza a operação descrita anteriormente e retorna um valor lógico, sinalizando se foi possível efetuar a procedimento. Conforme visto na imagem abaixo, verifica-se se foi passado uma dimensão válida para realizar a operação, aloca-se dinamicamente uma matriz ao ponteiro passado como parâmetro e posteriormente se realiza a soma através de uma estrutura de repetição duplamente encadeada com . Após isso, retorna-se verdadeiro se a condição inicial for falsa, senão se retorna falso sinalizando que não é possível realizar a soma com dimensão inválida. A imagem abaixo ilustra o código do item A.



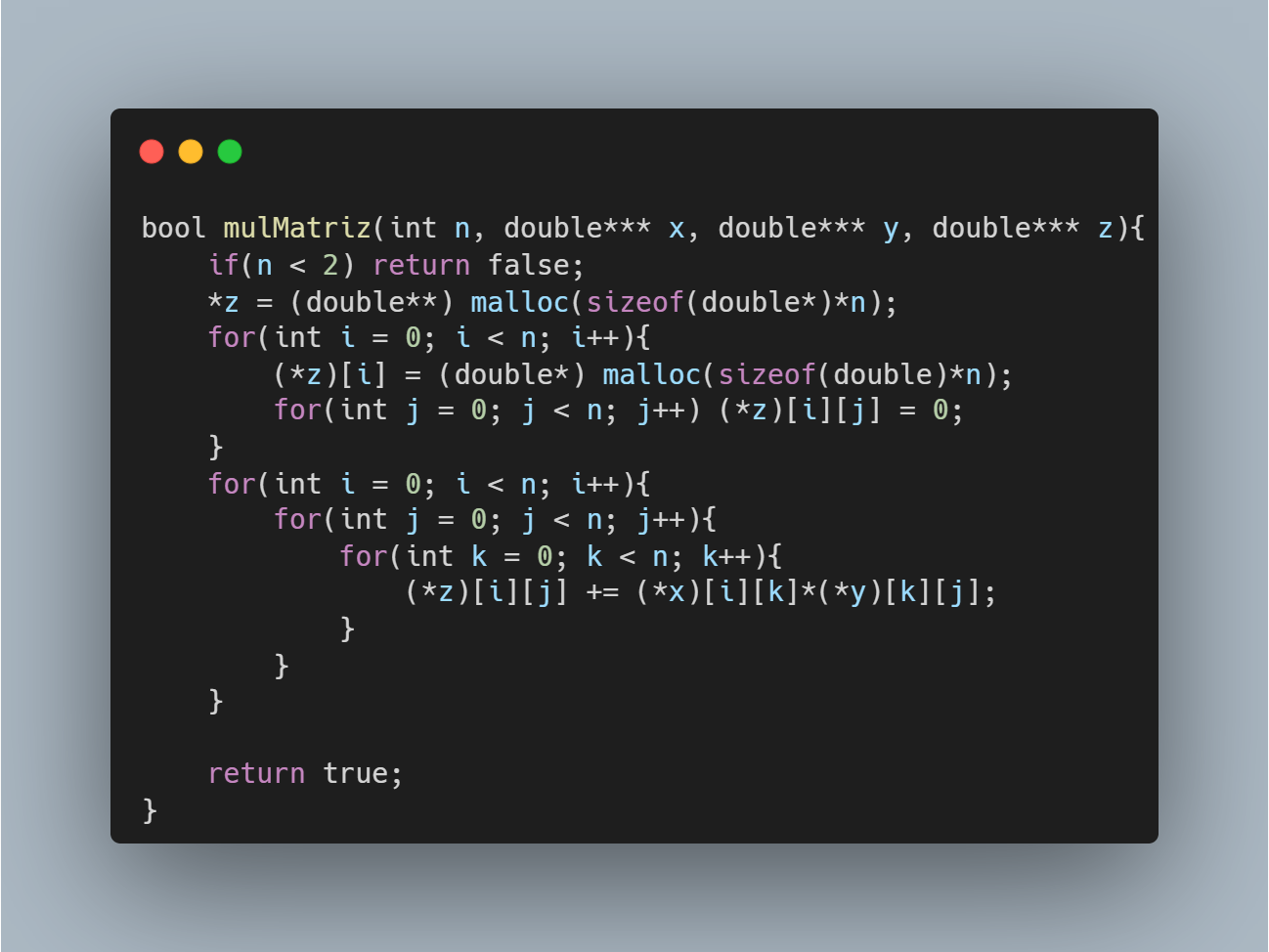
Ademais, o item B solicita que se produza uma função, o qual recebe como parâmetros um inteiro n, que representa a dimensão da matriz quadrada real , um ponteiro para a matriz real , uma constante real e um ponteiro que receberá uma matriz quadrada real resultante da operação da função. Nesse viés, a função multiplicará a matriz pela constante e atribuirá a matriz resultante no ponteiro passado como parâmetro, ou seja, .

Assim, foi feita a função do item B com a linguagem C, o qual se denomina “mulConstMatriz”, em que se realiza a operação descrita anteriormente e retorna um valor lógico, sinalizando se foi possível efetuar a procedimento. Conforme visto na imagem abaixo, verifica-se se foi passado uma dimensão válida para realizar a operação, aloca-se dinamicamente uma matriz ao ponteiro passado como parâmetro e posteriormente se realiza a multiplicação através de uma estrutura de repetição duplamente encadeada com . Após isso, retorna-se verdadeiro se a condição inicial for falsa, senão se retorna falso sinalizando que não é possível realizar a multiplicação com dimensão inválida. A imagem abaixo ilustra o código do item B.



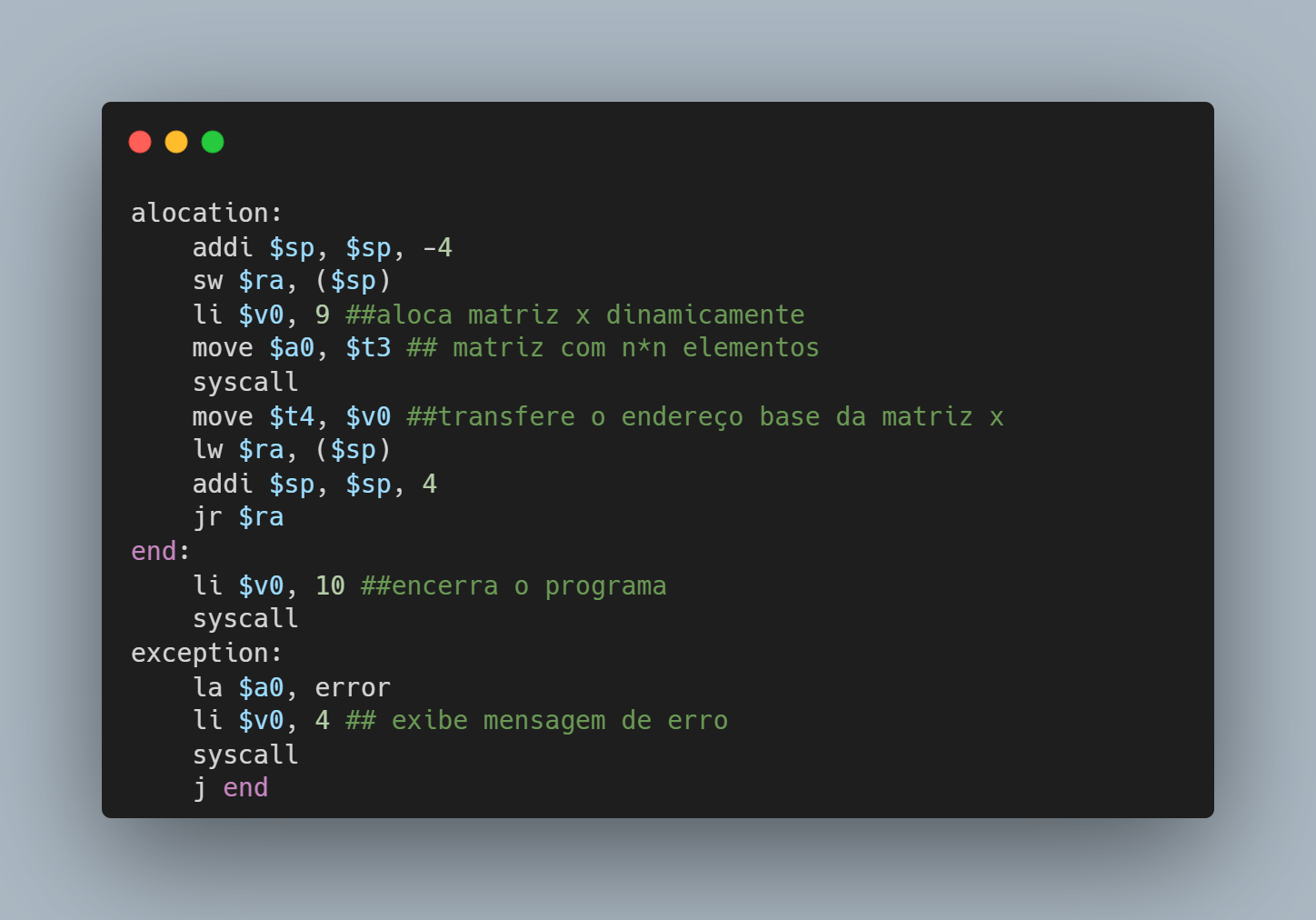
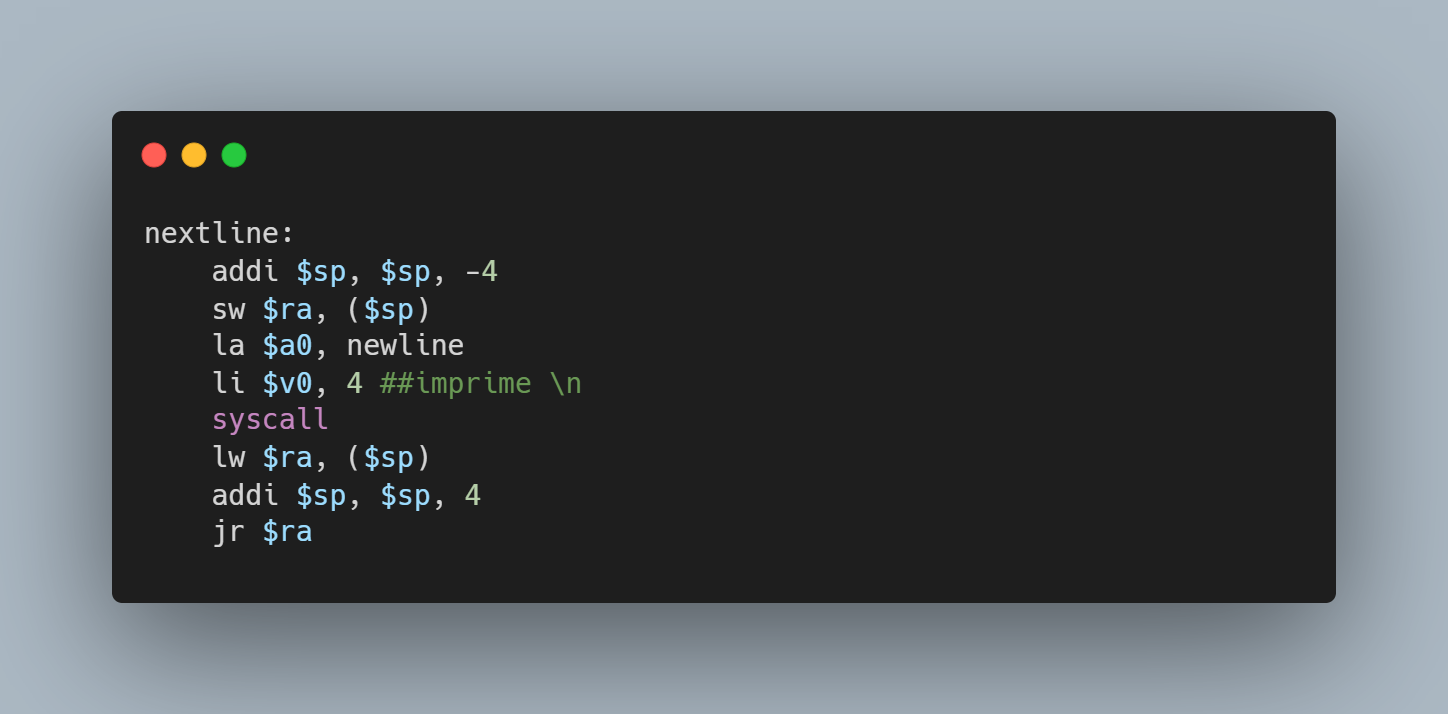
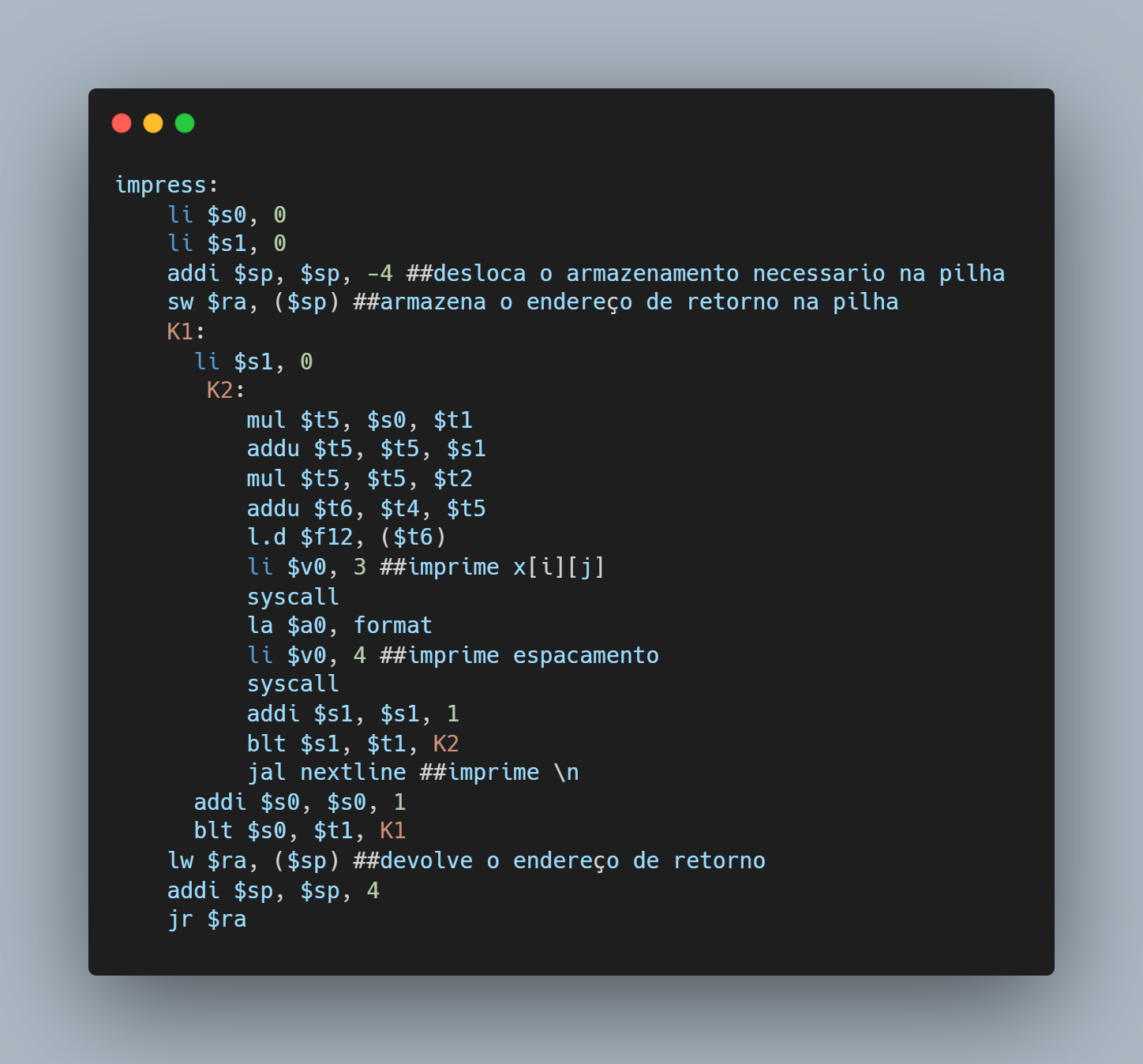
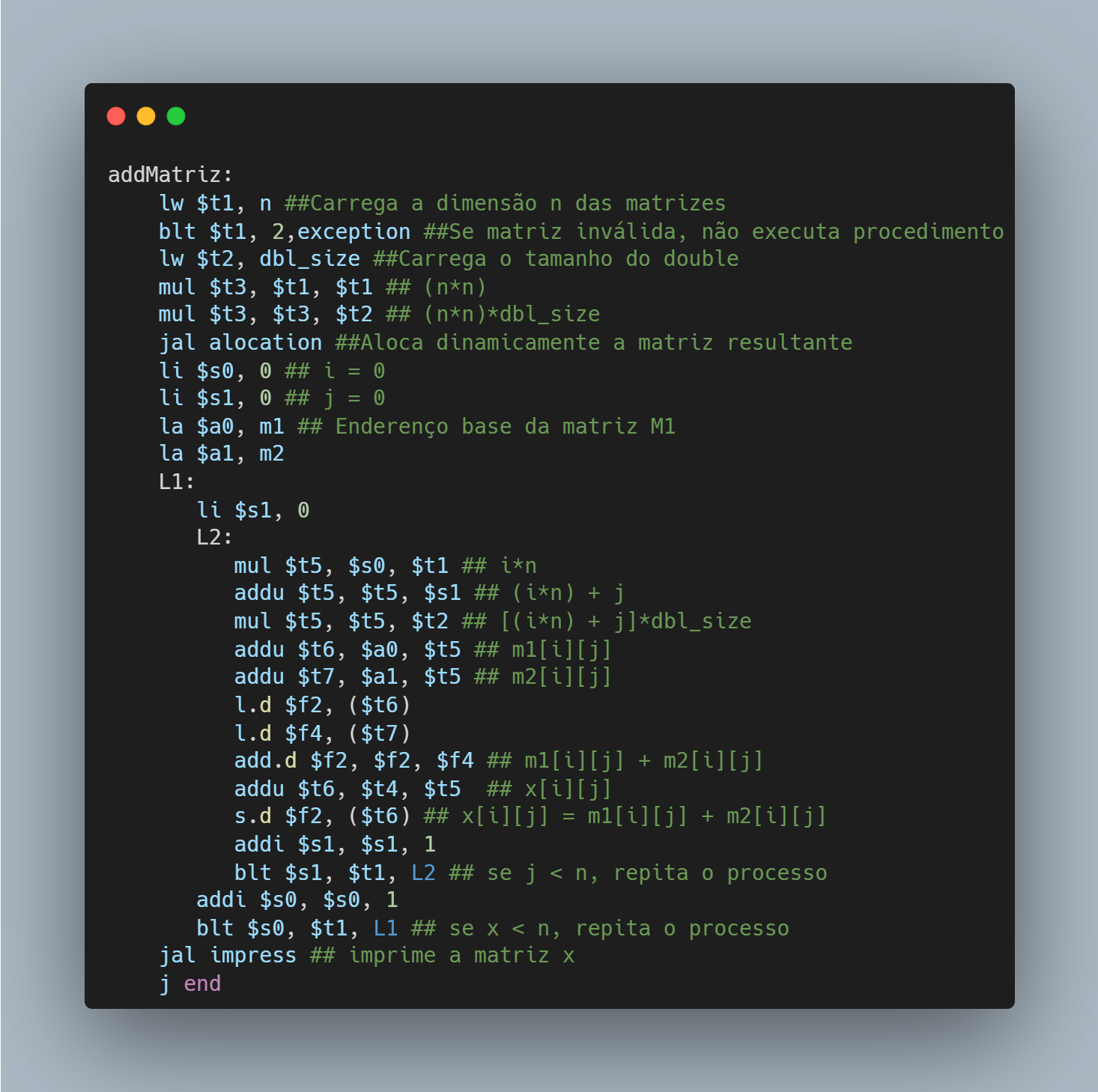
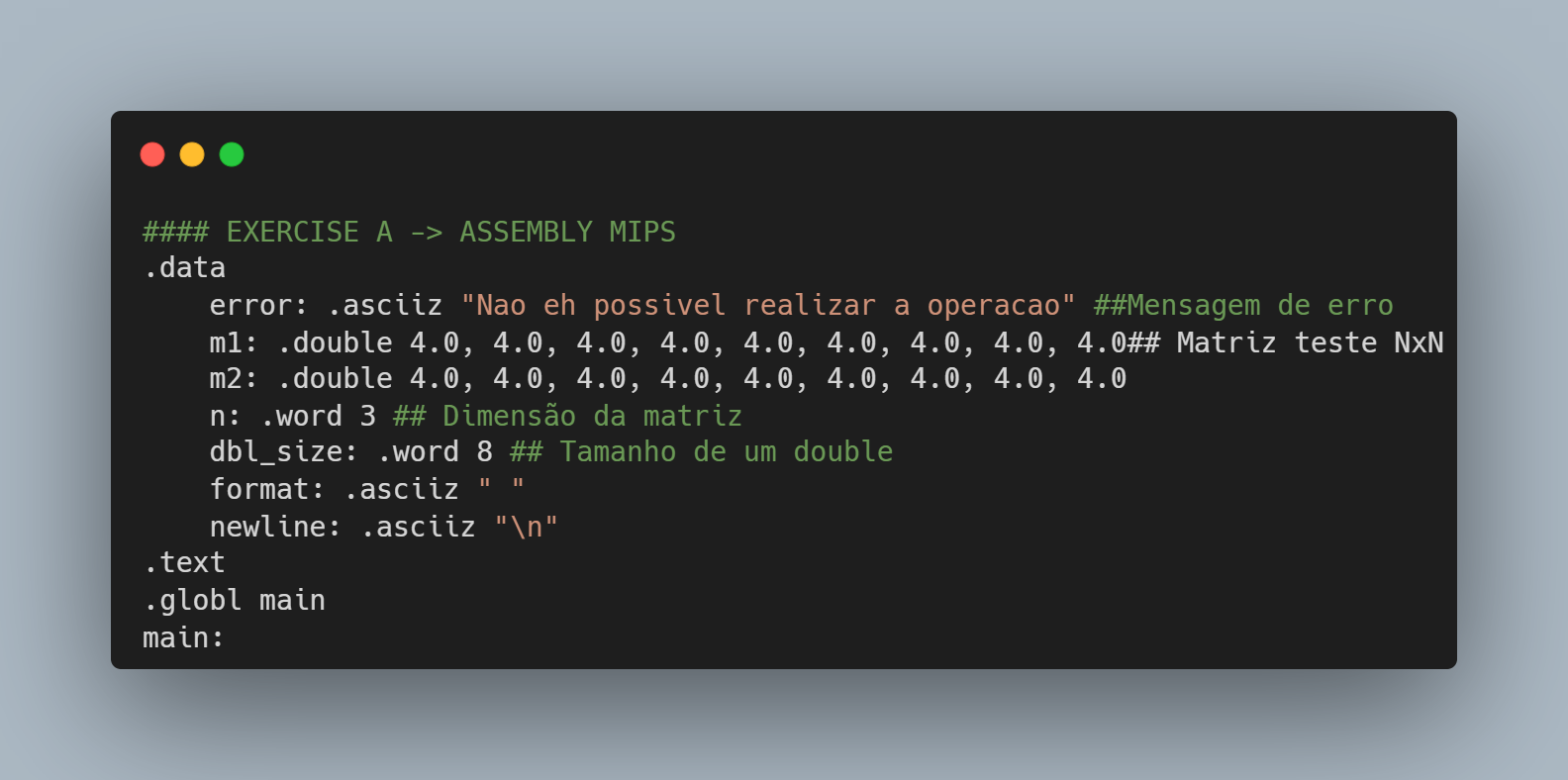
Outrossim, o item C solicita que se produza uma função, o qual recebe como parâmetros um inteiro n, que representa a dimensão da matriz quadrada real , dois ponteiros para as matrizes reais e um ponteiro que receberá uma matriz quadrada real resultante da operação da função. Nesse viés, a função multiplicará a matriz pela matriz e atribuirá a matriz resultante no ponteiro passado como parâmetro, ou seja, .

Dessa forma, foi efetuada a função do item C com a linguagem C, o qual se denomina “mulMatriz”, em que se realiza a operação descrita anteriormente e retorna um valor lógico, sinalizando se foi possível efetuar a procedimento. Consoante a imagem abaixo, verifica-se se foi passado uma dimensão válida para realizar a operação, aloca-se dinamicamente uma matriz ao ponteiro passado como parâmetro, inicializa a matriz alocada com valor zero e posteriormente se realiza a operação , que é a soma do valor anterior de e o produto e ,posteriormente, a atribuição deste resultado ao , através de uma estrutura de repetição triplamente encadeada com . Com isso, obtém-se e, assim, retorna-se verdadeiro se a condição inicial for falsa, senão se retorna falso sinalizando que não é possível realizar a multiplicação com dimensão inválida. A imagem abaixo ilustra o código do item C.



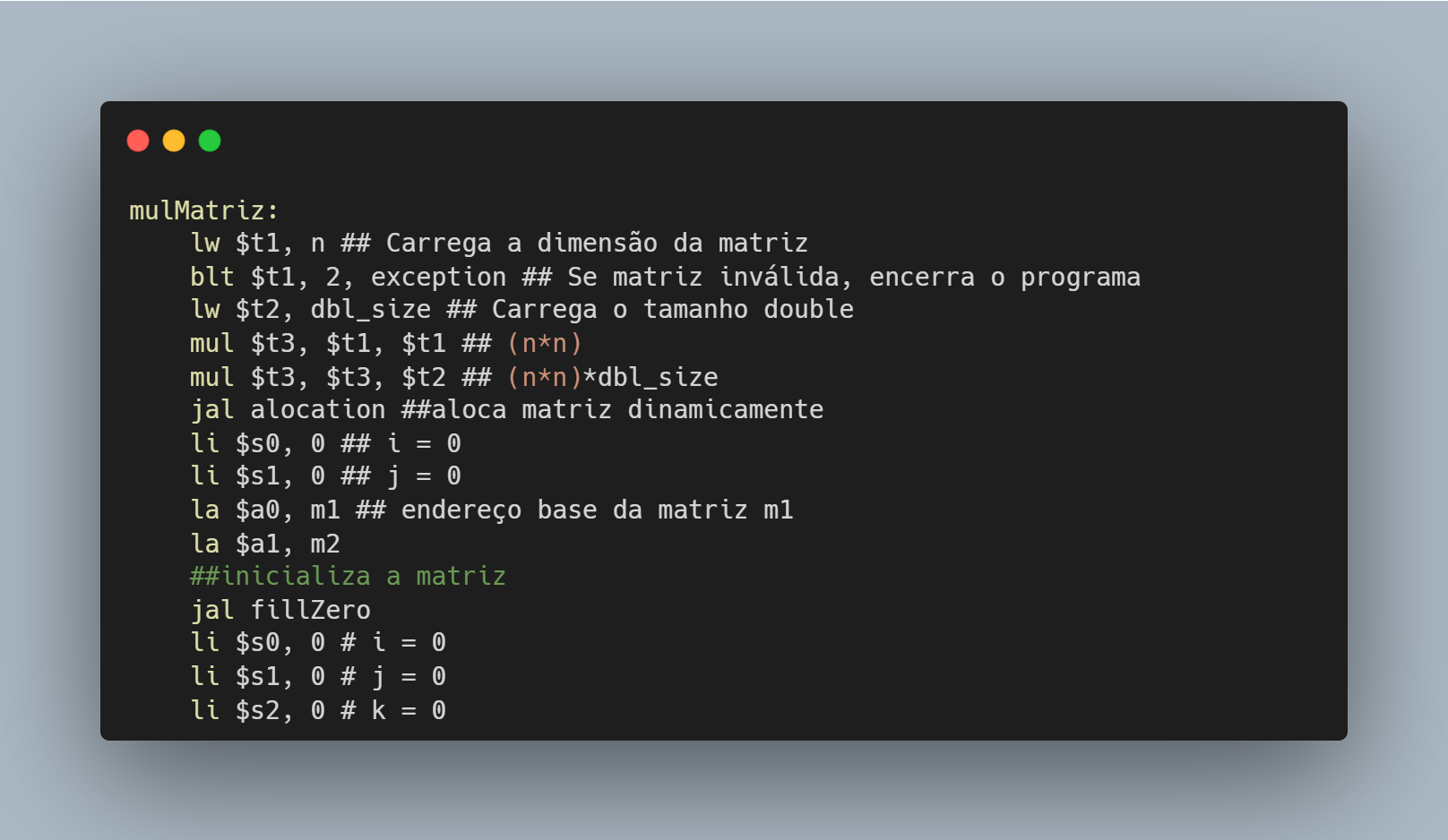
1. **Código em Assembly desenvolvido**

Nesta seção, é apresentado os códigos desenvolvidos em Assembly para solucionar os problemas propostos. Nesse sentido, as ilustrações abaixo mostram o programa feita para o exercício A.

****

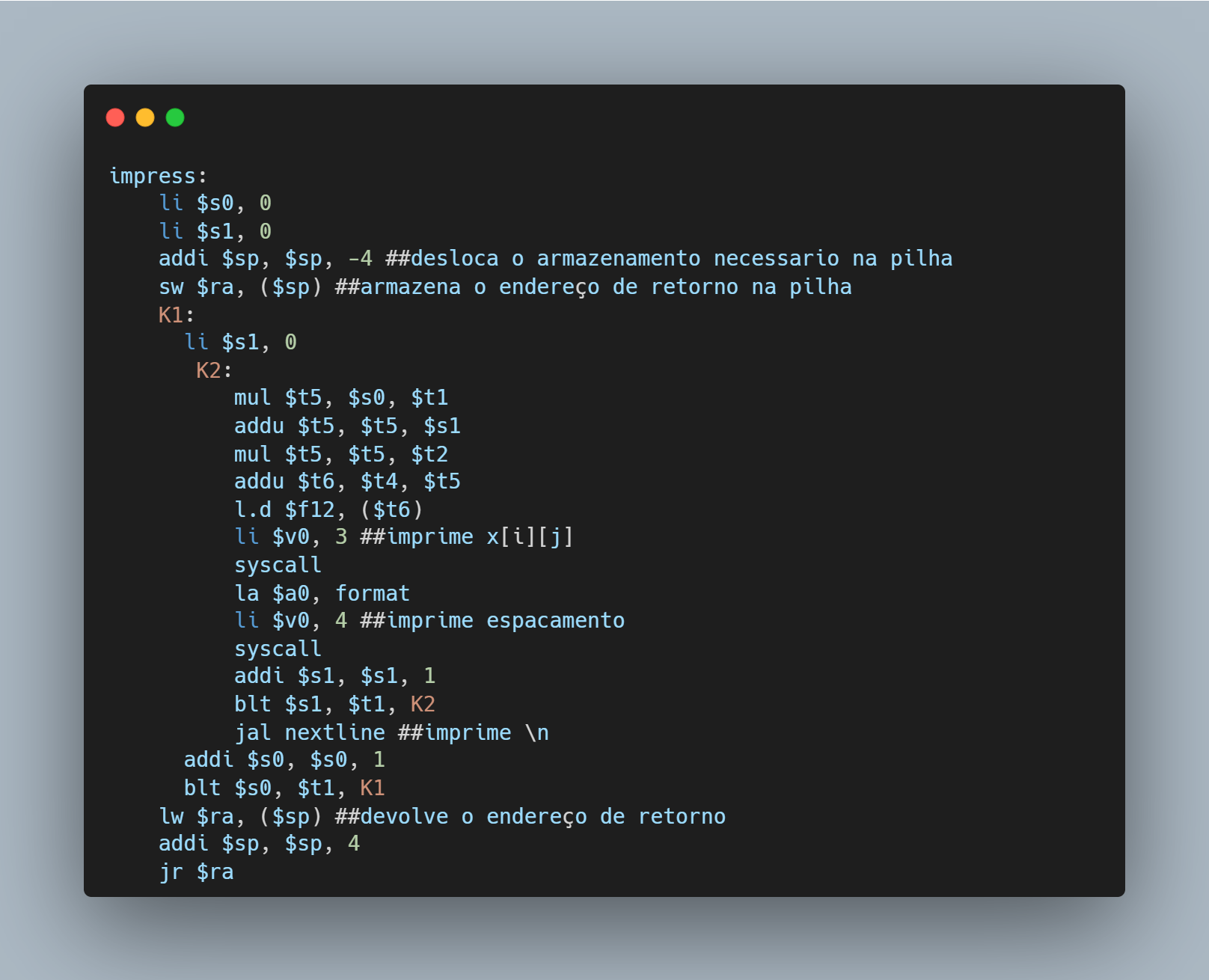
Outrossim, nas imagens a seguir se apresenta o código desenvolvido para resolver o problema C.













1. **Explicação detalhada das instruções utilizadas no código feito pela equipe**

A seguir, apresenta-se as instruções utilizadas da arquitetura MIPS para desenvolver os programas em Assembly, considerando o nível de detalhe em micro-operações.

* blt $t1, 2, exception: é uma pseudoinstrução que verifica se o valor do registrador é menor que o número passado e, em caso verdadeiro, se desvia para a label, senão continua para a instrução seguinte. Nesse caso, é uma pseudoinstrução com valor imediato.
* lw $t1, n: é uma instrução que carrega uma palavra (word) de 32 bits do endereço n ao registrador t1.
* mul $t3, $t1, $t1: é uma instrução que realiza a multiplicação entre os últimos registradores da instrução e atribui o resultado ao registrador t3.
* jal multiplication: é uma instrução que realiza o desvio incondicional à label multiplication e guarda o endereço contido em program counter no registrador ra (return address).
* li $s0, 0: é uma pseudoinstrução que atribui o valor imediato ao registrador passado.
* la $a0, m1: é uma pseudoinstrução que carrega o endereço de memória m1 para o registrador.
* addu $t6, $a0, $t5: é uma instrução que soma os valores sem sinal dos registradores a0 e t5 e atribui o resultado ao registrador t6.
* l.d $f2, ($t6): é uma pseudoinstrução que carrega o valor de ponto flutuante com precisão dupla ao registrador f2.
  1. **Instrução blt $t1, 2, exception**

Primeiramente, branch if less than (blt) é uma pseudoinstrução, ou seja, não é uma instrução nativa do repertório da arquitetura MIPS. Dessa forma, o assembler traduz a pseudoinstrução blt para uma combinação de instruções entre slt (Set on Less Than), bne (Branch if Not Equal) e a pseudoinstrução li (Load Immediate), no caso de blt com valor imediato. No caso do li, é feita a conversão para as instruções lui (Load Upper Immediate) e ori (OR Immediate) ou para addiu (Add Immediate Unsigned), caso o valor imediato seja de 32 bits ou 16 bits, respectivamente. Assim, a execução do blt imediato com valor 2 equivale à equação sequencial do addiu, slt e bne, respectivamente.

Então, logo abaixo estão as micro-operações envolvidas e o formato em cada instrução.

* addiu $at, $zero, 2



//Ciclo de Busca

A1: MAR <- PC;

A2: MBR <- Memory

PC <- PC + 4;

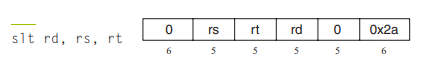
A3: IR <- MBR;

// Não há ciclo indireto

//Ciclo de Execução

A1: RT <- (RS) + IR(imm)

* slt $at, $t1, $at



//Ciclo de Busca

A1: MAR <- PC;

A2: MBR <- Memory

PC <- PC + 4;

A3: IR <- MBR;

// Não há ciclo indireto

//Ciclo de Execução

A1: if(RS < RT) Then RD <- 1

else RD <- 0

* bne $at, $zero, exception



//Ciclo de Busca

A1: MAR <- PC;

A2: MBR <- Memory

PC <- PC + 4;

A3: IR <- MBR;

// Não há ciclo indireto

//Ciclo de Execução

A1: if(RS != RT) Then PC <- PC + IR(offset)

* 1. **Instrução lw $t1, n**

****

//Ciclo de Busca

A1: MAR <- PC;

A2: MBR <- Memory

PC <- PC + 4;

A3: IR <- MBR;

// Não há ciclo indireto

//Ciclo de Execução

A1: MAR <- RS + IR(offset)

A2: MBR <- (Memory)

A3: RT <- (MBR)

* 1. **Instrução mul $t3, $t1, $t1**

****

//Ciclo de Busca

A1: MAR <- PC;

A2: MBR <- Memory

PC <- PC + 4;

A3: IR <- MBR;

// Não há ciclo indireto

//Ciclo de Execução

A1: RD <- (RS) \* (RT)

* 1. **Instrução jal multiplication**

****

* 1. **Instrução li $s0, 0**

//Ciclo de Busca

A1: MAR <- PC;

A2: MBR <- Memory

PC <- PC + 4;

A3: IR <- MBR;

// Não há ciclo indireto

//Ciclo de Execução

A1: RA <- PC

A2: PC <- {PC[31:28], IR(target) \* 4} //concatenação

Em primeiro lugar, li (Load Immediate) é uma pseudoinstrução, ou seja, não é uma instrução nativa do repertório do MIPS. Assim, é feita a tradução do li para addiu (Add Immediate Unsigned) ou para lui (Load Upper Immediate) e ori (OR Immediate), quando o valor imediato é de 16 bits ou 32 bits, respectivamente. Nesse caso, foi passado um valor de 16 bits, então haverá a execução da instrução addiu para realizar a funcionalidade do li.



//Ciclo de Busca

A1: MAR <- PC;

A2: MBR <- Memory

PC <- PC + 4;

A3: IR <- MBR;

// Não há ciclo indireto

//Ciclo de Execução

A1: RT <- (RS) + IR(imm)

* 1. **Instrução la $a0, m1**

Primeiramente, la (Load Address) é uma pseudoinstrução, ou seja, não é uma instrução nativa da arquitetura MIPS. Nesse sentido, se o endereço pode ser representado um número dentro de 16 bits, é utilizado a instrução addi (Add Immediate), caso contrário é usado as instruções lui (Load Upper Immediate) e ori (OR Immediate).

Para a primeira alternativa, temos:



//Ciclo de Busca

A1: MAR <- PC;

A2: MBR <- Memory

PC <- PC + 4;

A3: IR <- MBR;

// Não há ciclo indireto

//Ciclo de Execução

A1: RT <- (RS) + IR(imm);

Para a segunda alternativa, temos:



//Ciclo de Busca

A1: MAR <- PC;

A2: MBR <- Memory

PC <- PC + 4;

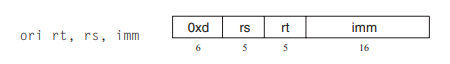
A3: IR <- MBR;

// Não há ciclo indireto

//Ciclo de Execução

A1: ALUOut <- IR(imm) << 16;//Coloca 16 bits zeros

A2: RT <- ALUOut;



//Ciclo de Busca

A1: MAR <- PC;

A2: MBR <- Memory

PC <- PC + 4;

A3: IR <- MBR;

// Não há ciclo indireto

//Ciclo de Execução

A1: IMM <- ZeroExtend(IR(imm)); //Estende com zeros

A2: ALUOut <- RS | IMM;

A3: RT <- ALUOut;

* 1. **Instrução addu $t6, $a0, $t5**

****

//Ciclo de Busca

A1: MAR <- PC;

A2: MBR <- Memory

PC <- PC + 4;

A3: IR <- MBR;

// Não há ciclo indireto

//Ciclo de Execução

A1: RD <- (RS) + (RT)

* 1. **Instrução l.d $f2, ($t6)**

Primeiramente, na arquitetura MIPS, os números de pontos flutuantes com precisão dupla são representados por 64 bits. Nesse sentido, deve-se armazenar os 32 bits mais significativos em um registrador e o restante em outro, pois os registradores suportam apenas 32 bits. Então, a pseudoinstrução l.d é convertida em dois lw (Load Word), onde f2 armazenará os bits mais significativos e f3 conterá os menos significativos. Dessa forma, executar l.d $f2, ($t6) é equivalente a realizar as instruções lw $f2, 0($t6) e lw $f3, 4($t6), para pegar os 32 bits mais significativos e menos significativos, respectivamente.

* lw $f2, 0($t6)



//Ciclo de Busca

A1: MAR <- PC;

A2: MBR <- Memory

PC <- PC + 4;

A3: IR <- MBR;

// Não há ciclo indireto

//Ciclo de Execução

A1: MAR <- RS +IR(offset)

A2: MBR <- (Memory)

A3: RT <- (MBR)

* lw $f3, 4($t6)



//Ciclo de Busca

A1: MAR <- PC;

A2: MBR <- Memory

PC <- PC + 4;

A3: IR <- MBR;

// Não há ciclo indireto

//Ciclo de Execução

A1: MAR <- RS +IR(offset)

A2: MBR <- (Memory)

A3: RT <- (MBR)