Prova com consulta. Duração: 1h30m. Segundo Mini-Teste

Grupo 1. Representações Intermédias de Baixo Nível (5 valores)

Considere o trecho de programa C e os tipos de variáveis e locais de armazenamento das mesmas indicados na tabela abaixo.

```
1. for(int i=0; i<N; i++) {
2.     A[i] = A[i]*c+d;
3. }
```

1.a) [2v] Indique para o trecho de código à esquerda, a LLIR (representação intermédia de baixo nível) baseada na representação intermédia em árvores de expressões apresentada nos slides das aulas teóricas;

1.b) [3v] Desenhe a LLIR para o trecho de código tendo por base árvores de expressões, mas considerando o processador *Jouette+* (cujo conjunto de instruções é apresentado em anexo).

Variável	Tipo	Armazenamento
A	int8 A[N] (array de inteiros (8-bit))	Endereço base do array armazenado na pilha em SP + 4
c	int c (variável escalar de 32-bit)	Variável armazenada no registo r2
d	int d (variável escalar de 32-bit)	Variável armazenada no registo r3
i	int i (variável escalar de 32-bit)	Variável armazenada no registo r4
N	int N (variável escalar de 32-bit)	Variável armazenada no registo r1

Grupo 2. Seleção de Instruções e Geração de Código (5 valores)

2.a) [2v] Realize a seleção de instruções usando o algoritmo Maximal Munch para cada subárvore na LLIR da alínea **1b)**, indicando a cobertura efetuada.

2.b) [2v] Indique a sequência de instruções geradas considerando a seleção de instruções obtida pelo algoritmo Maximal Munch.

2.c) [1v] Desenhe uma LLIR que permita gerar código mais eficiente do que a LLIR da alínea **1b)**, tendo por base árvores de expressões, mas considerando o processador *Jouette* + (cujo conjunto de instruções é apresentado em anexo), quando "N" representa uma constante de valor 1000.

Grupo 3. Alocação de Registos/Memória (6 valores)

No contexto da alocação de espaço na pilha para *arrays* locais a funções, pretende-se desenvolver um compilador que sempre que possível atribua o mesmo espaço de pilha a mais do que um array. No âmbito deste problema, responda às seguintes questões:

```
// Live-in={d}
// N e M representam constants inteiras

1.    int A[N];
2.    i=0;
3.    max = MIN_INT;
4.    loop1: if(i>=N) goto end1;
5.    A[i] = read();
6.    if(A[i] > max) max = A[i];
7.    i = i+1;
8.    goto loop1;
9.    end1:

10.   s = min(M,N);
11.   int B[s];
12.   i=0;
13.   loop2: if(i>=s) goto end2;
14.   B[i] = A[i]/max;
15.   i = i+1;
16.   goto loop2;
17.   end2:

18.   int C[s];
19.   i=0;
20.   loop3: if(i>=s) goto end3;
21.   C[i] = B[i]*d;
22.   i = i+1;
23.   goto loop3;
24.   end3:

// o array C é o único array usado a partir deste ponto
```

3.a) [2v] Descreva como poderia resolver o problema considerando o uso de *liveness analysis* (análise do tempo de vida);

3.b) [2v] Indique para cada linha numerada do trecho de código à esquerda, os conjuntos de *def* e *use* que usaria para efetuar *liveness analysis*.

3.c) [1v] Indique por inspeção direta do código à esquerda, o grafo de interferências que consideraria.

3.d) [1v] Descreva como poderia usar coloração de grafos e indique o resultado da coloração de grafos considerando o grafo da alínea anterior.

Grupo 4. Geral (4 valores)

No contexto de *register spilling*, os usos e definições das variáveis que são armazenadas em memória são traduzidos para leituras (loads) e escritas (stores) da/na memória. Quando existem vários usos de uma variável, sem existirem definições dessa variável entre esses usos, poderemos evitar alguns acessos à memória (loads). Explique de que forma é que alguns acessos à memória podem ser evitados, indicando as vantagens e desvantagens dessa abordagem, e uma heurística que permita decidir se cada uso é traduzido em leitura da memória ou não.

(Fim.)