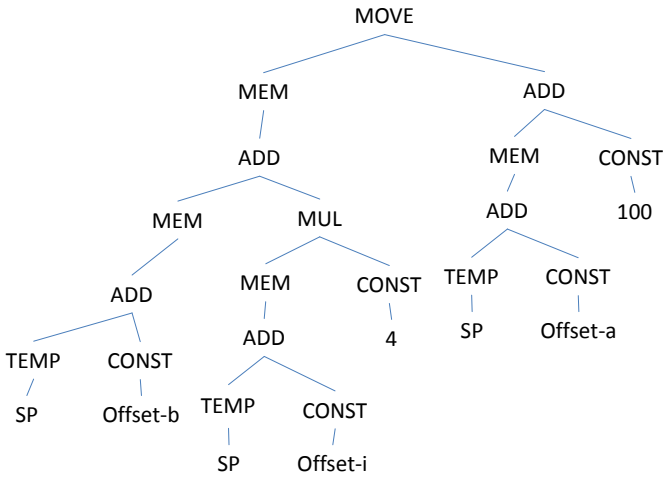
Prova com consulta. Duração: 1h30m. Primeiro Mini-Teste

Grupo 1. (9 valores)



Considere a representação intermédia para um trecho de código apresentada em baixo. Considere que *Offset-a*, *Offset-b* e *Offset-i* identificam constantes e SP (*stack pointer*) identifica o registo com o endereço da pilha.

1.a) [1v] Indique um possível trecho de código de uma linguagem de programação alto-nível (C ou Java) que pode estar na origem desta representação.

1.b) [2v] Apresente a representação intermédia designada por árvore de expressões equivalente à representação intermédia apresentada, seguindo o formato utilizado nos slides da disciplina.

1.c) [3v] Considerando o conjunto de instruções apresentado em (*), utilize o algoritmo *Maximal Munch* para seleccionar as instruções, indicando também a sequência de instruções resultante. Indique na árvore a cobertura da mesma pelas árvores de padrões de instruções seleccionadas.

1.d) [3v] Considerando o conjunto de instruções apresentado em (*), use programação dinâmica para seleccionar as instruções para o exemplo, indicando se resulta num número menor de instruções. Explique a razão caso resulte num número menor de instruções. Caso resulte num número igual de instruções, apresente uma ou mais instruções novas que mostrem como o uso de programação dinâmica pode dar melhores resultados do que o uso do *Maximal Munch*.

(*) Conjunto de instruções (considere o mesmo custo para todas as instruções) de um processador hipotético designado por *Jouette*:

ADD rd = rs1 + rs2 ADDI rd = rs + c	SUB rd = rs1 − rs2 SUBI rd = rs – c MUL rd = rs1* rs2 DIV rd = rs1/rs2	LOAD rd = M[rs + c] STORE M[rs1 + c] = rs2 MOVEM M[rs1] = M[rs2]

Em que *rd* e *rs* identificam registos de 32 bits do processador (de *r0* a *r31*, tendo *r0* o valor 0), *c* identifica um literal e *M[]* indica o acesso à memória.

As correspondentes árvores de padrões de instruções são as seguintes (note que + equivale a ADD, e * a MUL):

Name	*Effect*	*Trees*
—	r_i	TEMP
ADD	$r_i \leftarrow r_j + r_k$	+
MUL	$r_i \leftarrow r_j \times r_k$	*
SUB	$r_i \leftarrow r_j - r_k$	-
DIV	$r_i \leftarrow r_j / r_k$	/
ADDI	$r_i \leftarrow r_j + c$	+ CONST
SUBI	$r_i \leftarrow r_j - c$	- CONST
LOAD	$r_i \leftarrow M[r_j + c]$	MEM + CONST

STORE	$M[r_j + c] \leftarrow r_i$	MOVE MEM + CONST CONST
MOVEM	$M[r_j] \leftarrow M[r_i]$	MOVE MEM MEM

Grupo 2. (11 valores)

Considere o seguinte código. Note que todas as variáveis, *m, v, n, r,* e *x*, armazenam inteiros representados por 32 bits.

```
1.          m ← 0
2.          v ← 0
3.   lab3:  if v ≥ n goto lab2
4.          r ← v
5.   lab5:  if r < n goto lab1
6.          v ← v + 1
7.          goto lab3
8.   lab1:  x ← M[r]
9.          if x ≤ m goto lab4
10.         m ← x
11.  lab4:  r ← r + 1
12.         goto lab5
13.  lab2:  return m
```

2.a) [1v] Apresente o grafo de fluxo de controlo (CFG: *Control-Flow Graph*) para o código apresentado.

2.b) [2v] Utilize análise de fluxo de dados para determinar o tempo de vida das variáveis e indique o grafo de interferências resultante. Apresente as iterações necessárias para a análise de fluxo de dados apresentando os respetivos conjuntos de *def*, *use*, *in*, e *out*.

2.c) [1v] Comente a seguinte afirmação: "o uso de *register coalescing* permite reduzir o número de registos a utilizar para armazenar as variáveis do código apresentado".

2.d) [1v] Indique o número mínimo de registos necessário para armazenar as variáveis sem ter de fazer *register spilling*. Justifique a resposta.

2.e) [2v] Apresente uma possível alocação de registos a variáveis utilizando o algoritmo de coloração de grafos explicado nas aulas e supondo a utilização de 4 registos de 32 bits (*r1*, *r2*, *r3*, e *r4*). Apresente o conteúdo da pilha após simplificação do grafo de interferências. No caso de ter de fazer *register spilling* indique o critério que usou para a selecção de variáveis e apresente apenas o resultado da primeira coloração de grafos (i.e., antes de repetir o processo).

2.f) [1v] Apresente o código resultante após a primeira coloração de grafos da alocação de registos realizada na alínea anterior. Caso tenha sido necessário fazer *register spilling*, considere a existência das instruções R? = MEM[SP+const] (*load* para R? de um valor da posição de memória dada pela soma de uma constante com o valor de SP) e de MEM[SP+const] = R? (*store* do valor de R? na posição de memória dada pela soma de uma constante com o valor em SP). Considere que o valor de SP foi previamente definido. Note também que não necessita de incluir no código solicitado a alocação e dealocação de espaço na pilha para armazenar variáveis.

2.g) [3v] Indique e descreva sucintamente as otimizações que podem ser utilizadas para melhorar a alocação de registos a variáveis. Descreva como e quando é que cada uma dessas otimizações pode ser efectuada.

(Fim.)