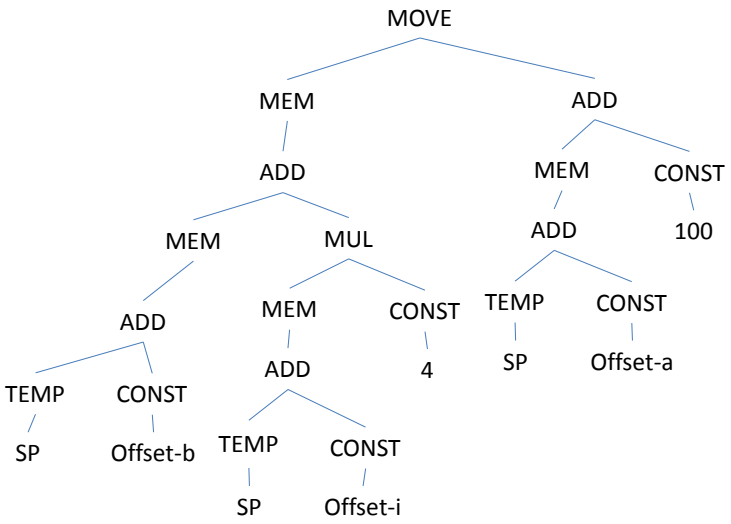
Prova com consulta. Duração: 1h30m. Primeiro Mini-Teste

Grupo 1. (9 valores)

Considere a representação intermédia para um trecho de código indicada 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.



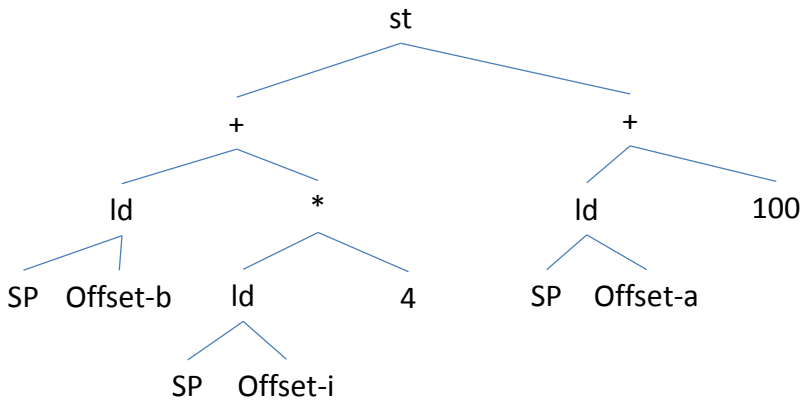
1.a) [1v] Indique o possível trecho de código de uma linguagem de programação alto-nível (C, por exemplo) que poderá ter originado esta representação.

b[i]=a+100;
- b (endereço base do array b), a, e i guardados na pilha
- a uma variável de tipo int
- b um array de int's
- i uma variável de tipo int

[*(b+i*4)=a+100;]

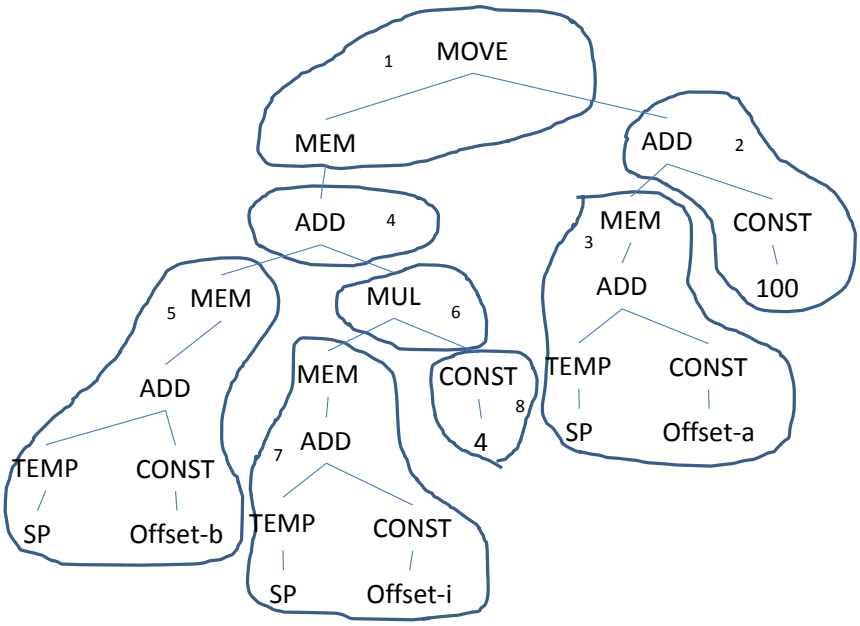
1.b) [2v] Apresente a representação intermédia utilizada nos slides da disciplina, designada por árvore de expressões, equivalente à representação intermédia apresentada.

Possível árvore de expressões:



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

Cobertura:



Instruções:
3. R1 = M[SP+Offset-a] // LOAD
2. R2 = R1 + 100 // ADDI
7. R3 = M[SP+Offset-i] // LOAD
8. R4 = R0 + 4 // ADDI
6. R5 = R3 * R4 // MUL
5. R6 = M[SP+Offset-b] // LOAD
4. R7 = R6 + R5 // ADD
1. M[R7+0] = R2 // STORE

1.d) [3v] Considerando o conjunto de instruções apresentado em (*), indique se o uso de programação dinâmica para seleccionar as instruções para o exemplo resulta num número menor de instruções. No caso de resultar num número menor indique a razão. No caso de resultar num número igual de instruções, apresente possíveis novas instruções que fariam com que o uso de programação dinâmica pudesse resultar num número menor de instruções do que o uso do *Maximal Munch*.

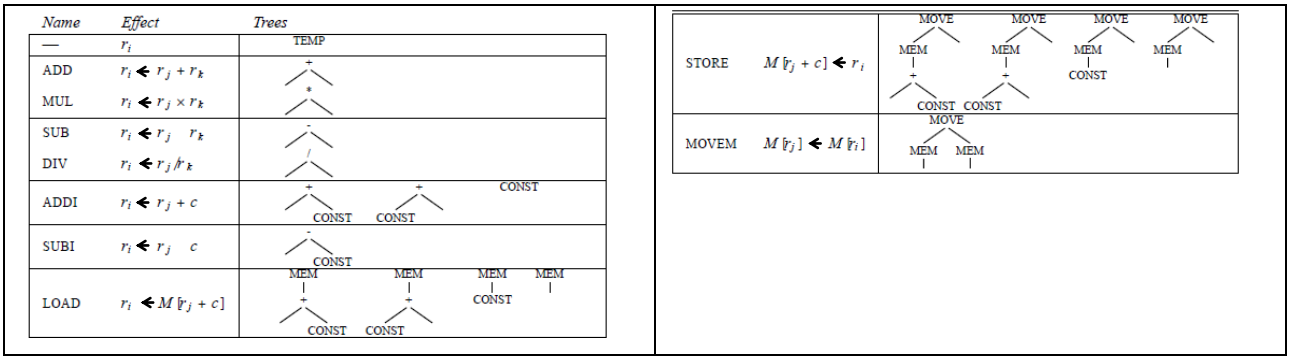
O uso de programação dinâmica para este exemplo resulta no mesmo número de instruções do que o uso do *Maximal Munch.*
A existência da instrução Rd=M[Rs1]+Rs2 poderia implicar uma instrução a menos usando programação dinâmica em relação ao uso do *Maximal Munch*.

(*) 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-bit 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, * a MUL):

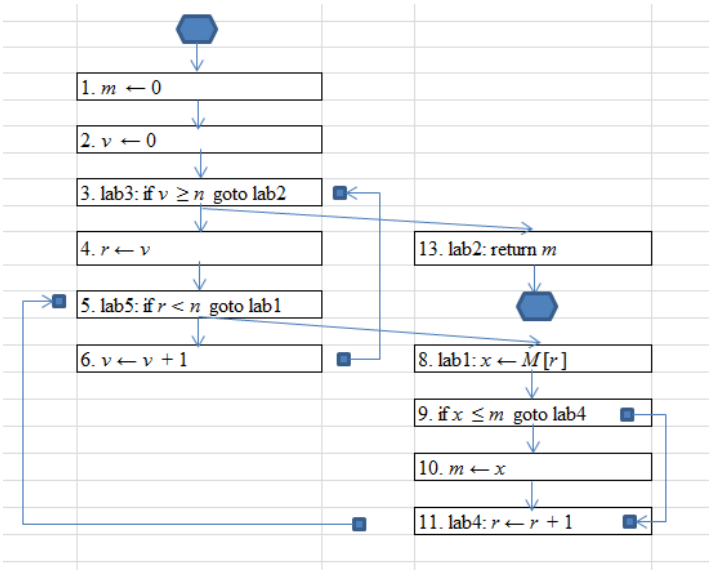


Grupo 2. (11 valores)

Considere o código seguinte. 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.

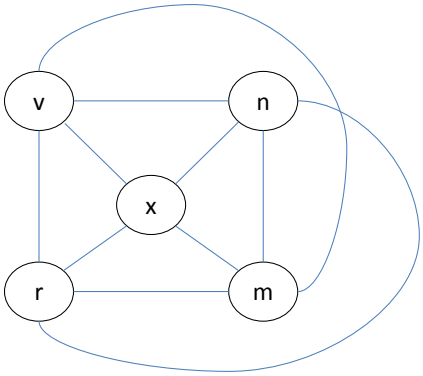


2.b) [2v] Utilize análise de fluxo de dados para determinar o tempo de vida das variáveis e indique o grafo de interferências resultante. Apresente as iterações necessárias para a análise de fluxo de dados apresentando os respetivos conjuntos de *def*, *use*, *in*, e *out*.

Resultado da análise do tempo de vida usando análise de fluxo de dados em sentido inverso ao fluxo (de 13 para 1):

					1ª iteração		2ª iteração		3ª iteração	
			def	use	out	in	out	in	out	in
	1. $m \leftarrow 0$	1	m		n,m	n	n,m	n	n,m	n
	2. $v \leftarrow 0$	2	v		v,n,m	n,m	v,n,m	n,m	v,n,m	n,m
	3. lab3: if $v \geq n$ goto lab2	3		v,n	v,n,m	v,n,m	v,n,m	v,n,m	v,n,m	v,n,m
	4. $r \leftarrow v$	4	r	v	r,n,v,m	v,n,m	v,n,m,r	v,n,m	v,n,m,r	v,n,m
	5. lab5: if $r < n$ goto lab1	5		r,n	v,r,m	r,n,v,m	v,n,m,r	v,n,m,r	v,n,m,r	v,n,m,r
	6. $v \leftarrow v + 1$	6	v	v		v	v,n,m	v,n,m	v,n,m	v,n,m
	7. goto lab3									
	8. lab1: $x \leftarrow M[r]$	8	x	r	x,m,r	r,m	x,r,n,v,m	r,n,v,m	x,r,n,v,m	r,n,v,m
	9. if $x \leq m$ goto lab4	9		x,m	x,r	x,m,r	x,r,n,v,m	x,r,n,v,m	x,r,n,v,m	x,r,n,v,m
	10. $m \leftarrow x$	10	m	x	r	x,r	r,n,v,m	x,r,n,v	r,n,v,m	x,r,n,v
	11. lab4: $r \leftarrow r + 1$	11	r	r		r	r,n,v,m	r,n,v,m	r,n,v,m	r,n,v,m
	12. goto lab5									
	13. lab2: return m	13		m		m		m		m

Grafo de interferências resultante:



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".

A afirmação anterior é falsa. As duas instruções de *move* (r = v e m = x) no trecho de código apresentado usam variáveis com interferências em termos de tempo de vida. A análise do referido código revela que não se pode fazer *register coalescing* para essas variáveis pois tal facto altera a funcionalidade original do código.

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

Como as 5 variáveis estão todas vivas para pelo menos uma instrução do trecho de código, é necessário um mínimo de 5 registos para evitar *register spilling*.

2.e) [3v] 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.

Possível conteúdo da pilha:
(topo) m-r- x-v-n (may spill) (fundo)

A variável n foi escolhida para spilling tendo por base o menor número de def e use.

Possível resultado da primeira coloração do grafo:
r1 → {m}, r2 → {r}, r3→ {x}, r4 → {v}, spill n

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 em R? de um valor dado pela 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.

Código resultante considerando que o valor de n é guardado na posição da pilha dada por SP+4.
No código em baixo, as variáveis m, r, x, e v devem ser substituídas pelos respetivos registos:
r1 → {m}, r2 → {r}, r3→ {x}, r4 → {v}

```
                Previous definitions of n (n=...), should be substituted by M[SP+4] =...
1.              m ← 0
2.              v ← 0
3.      lab3:   raux1 = M[SP+4] // load value of n
                if v ≥ raux1 goto lab2
4.              r ← v
5.      lab5:   raux2 = M[SP+4] // load value of n
                if r < raux2 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.g) [3v] Indique e descreva sucintamente as possíveis otimizações que podem ser utilizadas para melhorar a alocação de registos a variáveis.

Três otimizações possíveis (Nota: sem as descrições solicitadas na pergunta):

- Reordenação das instruções (de acordo com as dependências entre instruções);
- Divisão dos segmentos de tempos de vida que possam existir para a mesma variável;
- Utilização de *register coalescing*.

(Fim.)