EIC0028-2S – COMPILADORES (CONT.)

Teste - 6 de junho de 2012 *Duração total (partes I + II): 1 hora e 30 minutos*

PARTE II (11 valores)

1. Considere a seguinte representação intermédia de um trecho de programa, onde r1, r2, ..., r8 representam variáveis de 32-bit.

```
live-in={}
1.    r1 = 1028;
2.    r2 = MEM(r1);
3.    r3 = r1 * r2;
4.    r4 = 256;
5.    r5 = r4 – r2;
6.    r6 = 128;
7.    r7 = r5 * r6;
8.    r8 = r7 – r3;
9.    MEM(r1)  = r8;
live-out={}
```

1.1 [2 val] Determine os conjuntos de *live-in* e de *live-out* para **cada uma das instruções** utilizando análise de fluxo de dados. Quantas iterações foram necessárias?

Análise do tempo de vida por ordem inversa ao fluxo de instruções:

Inst.	use	def	out	in	out	in
9	1,8			1,8		1,8
8	7,3	8	1,8	1,7,3	1,8	1,7,3
7	5,6	7	1,7,3	1,3,5,6	1,7,3	1,3,5,6
6		6	1,3,5,6	1,3,5	1,3,5,6	1,3,5
5	4,2	5	1,3,5	1,2,3,4	1,3,5	1,2,3,4
4		4	1,2,3,4	1,2,3	1,2,3,4	1,2,3
3	1,2	3	1,2,3	1,2	1,2,3	1,2
2	1	2	1,2	1	1,2	1
1		1	1		1	

Nota: para abreviar omitiu-se o "r" como prefixo dos números nas colunas use, def, in, e out.

Conjuntos de live-in na última coluna e de live-out na penúltima coluna.

Foram necessárias duas iterações.

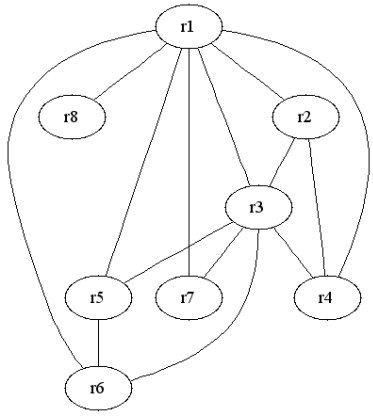
[a utilização do fluxo direto para determinar o tempo de vida das variáveis por análise de fluxo de dados deverá ser penalizada: -20% do valor da pergunta]

1.2 [2 val] Realize a alocação de registos utilizando o algoritmo de coloração de grafos apresentado nas aulas teóricas, considerando a utilização de **apenas 3 registos** de 32-bit representados por, $1, $2 e $3.

[nota 1: apresente apenas os resultados da **primeira iteração** da alocação de registos caso sejam necessárias várias iterações]

[nota 2: deve apresentar o conteúdo da pilha de variáveis imediatamente antes de começar a colorir o grafo, as variáveis atribuídas a cada um dos registos, e as variáveis para as quais tiver de fazer *register spilling*]

*Grafo de interferências resultante dos conjuntos de **in** e de **out** obtidos em 1.1:*



Pilha de variáveis (K=3):

r6
r5
r3
r4
r2
r1 may-spill
r7
r8

Possível alocação de registos:
$1→{r4, r6, r7, r8}
$2→{r5, r2}
$3→{r3}
spill r1

1.3 [1 val] Apresente a representação intermédia após a primeira iteração da alocação de registos realizada na alínea anterior.

Possível código depois tendo em conta os resultados da primeira iteração da alocação de registos:

live-in={}
1. *r1 = 1028;*
2. *MEM[r1-addr] = r1; // Exemplo: r1-addr = $sp-offset-r1*
3. *t1 = MEM[r1-addr];*
4. *$2 = MEM(t1);*
5. *t2 = MEM[r1-addr];*
6. *$3 = t2 * $2;*
7. *$1 = 256;*
8. *$2 = $1 – $2;*
9. *$1 = 128;*

*10. $1 = $2 * $1;*
11. $1 = $1 – $3;
12. t3 = MEM[r1-addr];
13. MEM(t3) = r8;
live-out={}

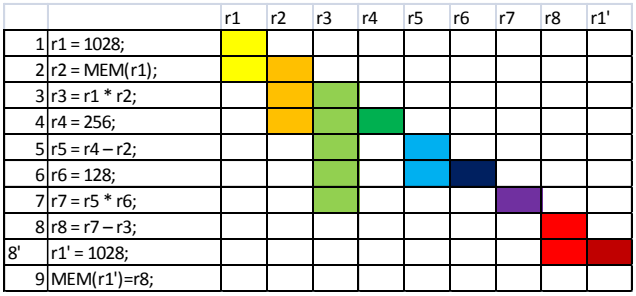
Nota: apresenta-se de seguida um possível trecho de código que evitaria os loads das linhas 3 à 5:

1. *r1 = 1028; // Exemplo: r1-addr = $sp-offset-r1*
2. *MEM[r1-addr] = r1;*
3. *$2 = MEM(r1);*
4. *$3 = r1 * $2;*

1.4 [2 val] Comente , sucintamente, as possíveis vantagens/desvantagens obtidas com as modificações da representação intermédia indicadas a cinzento no código abaixo.

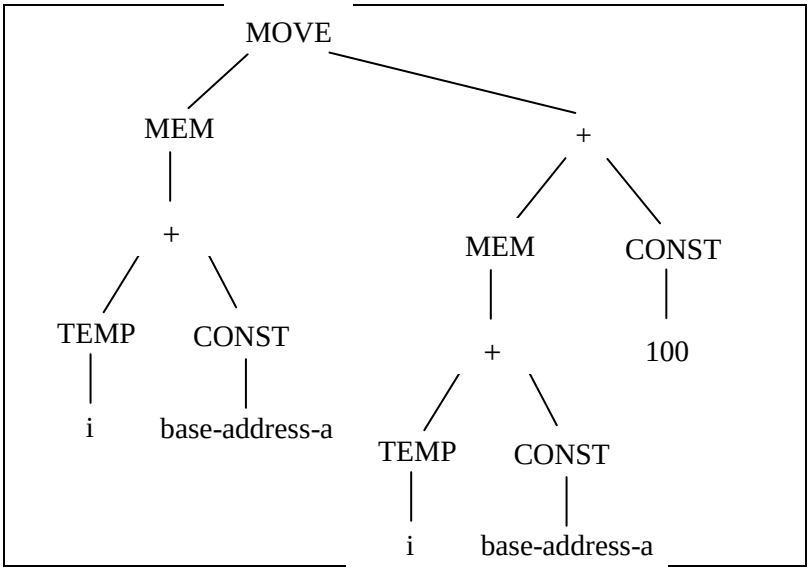
```
1.   r1 = 1028;
2.   r2 = MEM(r1);
3.   r3 = r1 * r2;
4.   r4 = 256;
5.   r5 = r4 – r2;
6.   r6 = 128;
7.   r7 = r5 * r6;
8.   r8 = r7 – r3;
8'.    r1' = 1028;
9.   MEM(r1')  = r8;
```

As modificações permitem a alocação das variáveis a 3 registos como se pode ver pelo tempo de vida das variáveis:



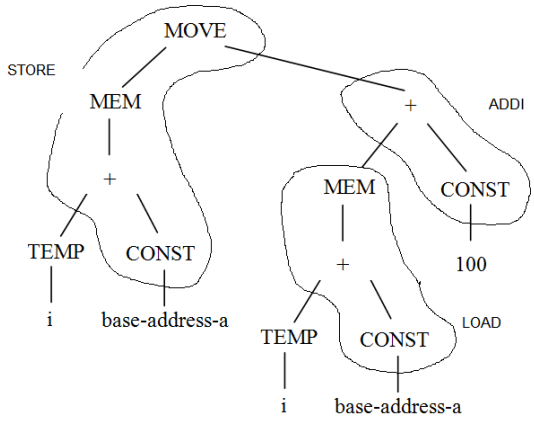
No entanto, é adicionada mais uma instrução ao código o que pode trazer a desvantagem de um aumento do número de ciclos de relógio para executar o código modificado vs. o número de ciclos de relógio para executar o código original (sem register spilling). Note-se, contudo, que a necessidade de register spilling no código original implica mais ciclos de relógio do que os que resultam destas modificações.

2. Suponha que se pretende gerar código para um processador hipotético cujas instruções são apresentadas em baixo(*). Considere o exemplo dado pela representação intermédia de baixo-nível apresentada de seguida:



2.1 [2 val] Considerando que *base-address-a* identifica uma constante com um endereço de memória, *i* uma variável armazenada no registo *r4*, e o conjunto de instruções apresentado em baixo, utilize o algoritmo *Maximal Munch* para seleccionar as instruções e indique a sequência de instruções resultante. [nota: indique na representação intermédia a cobertura da mesma pelas árvores de padrões de instruções]

Representação intermédia com a cobertura da mesma pelas árvores de padrões de instruções resultante da aplicação do algoritmo Maximal Munch para seleccionar as instruções:



Sequência de instruções resultante das instruções seleccionadas:
LOAD r3 = M[r4 + base-address-a]
ADDI r3 = r3 + 100
STORE M[r4 + base-address-a] = r3
[em vez de r3 pode ser utilizado um outro registo, à exceção do r0 e do r4]

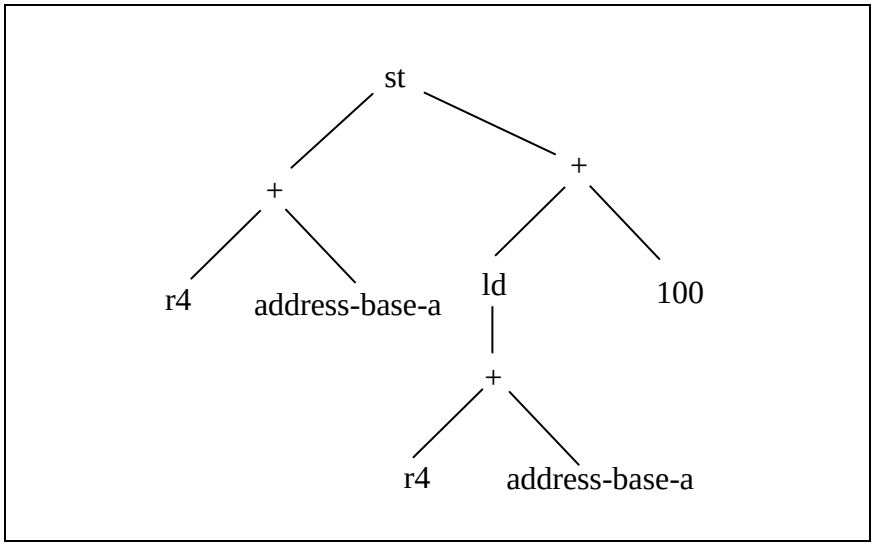
2.2 [1 val] Indique o possível trecho de código de uma linguagem de programação alto-nível (C, por exemplo) que poderá ter originado esta representação.

*int *a = (int *) base-address-a;*
**(a+i) = *(a+i) + 100;*

Ou, considerando que "a" identifica um array com inteiros representados por um byte e que o endereço base de a é dado por "base-address-a":
a[i] = a[i] + 100;

2.3 [1 val] Desenhe a representação intermédia utilizada nos slides da disciplina, designada por árvore de expressões, equivalente à representação intermédia apresentada.

Possível representação intermédia de baixo-nível baseada em árvore de expressões:



Nota: em vez de r4 poder-se-ia usar i e ser o descritor de i (e.g., na tabela de símbolos) a identificar que a variável i é armazenada no registo r4.

(*) As instruções do processador incluem:

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: