Correcção Teste Arquitectura Computadores

Pergunta 1:

Na instrução 8, o valor de %edx é reduzido em 4 unidades. Na instrução 9, verificamos que caso esta subtracção por 4 tenha dado 0, o salto não é tomado, saindo-se assim do ciclo. Por isto, concluímos que é o valor em %edx que controla o ciclo. %edx é inicializado a 200 e fica reduzido em 4 unidades em cada iteração. Sabemos também que o ciclo parará quando o seu valor for 0, logo o ciclo irá ser executado 200/4=50 vezes.

Instrução	Nº Execuções	CPI	#CC
1	1	1	1
2	1	1	1
3	50	3	150
4	50	3	150
5	50	1	50
6	50	3	150
7	50	1	50
8	50	1	50
9	50	2	100
		TOTAL =	**702 ciclos**

T_{exec}= #I * CPI * (1/f) = #CC * (1/f) = $702/10^9$ = 702 ns

Pergunta 2:

Na expressão T_{exe} =*#I x CPI x* T_{cc} apenas o T_{cc} não depende do compilador mas sim do *hardware* da máquina.

O compilador tem uma influência directa tanto no número de instruções como no CPI do programa, uma vez que é o responsável pelo nº de instruções geradas bem como pelo tipo de instruções geradas (instruções mais complexas terão CPI mais elevado, o que tenderá a elevar o CPI do programa).

Posto isto, há essencialmente duas formas simples pelas quais as opções de optimização poderão aumentar o desempenho: ou diminuindo o número de instruções, removendo instruções desnecessárias e/ou redundantes ou diminuindo o CPI do programa, através da utilização de instruções mais simples, eventualmente tentando substituir instruções que dependam de acessos à memória por outras que trabalhem com operandos em registos.

Como visto nas aulas práticas, geralmente o compilador pode optar por uma solução híbrida, em que de uma compilação –O0 para uma compilação -O3 o número de instruções é diminuído radicalmente, no entanto o CPI do programa também aumenta ligeiramente, o que sugere que o programa passa a ter menos instruções, mas mais complexas. Mesmo com o aumento do CPI, a redução do número de instruções é geralmente suficiente para que ainda assim se verifique um ganho bastante razoável no tempo de execução.

Pergunta 3:

O aumento do grau de associatividade da memória permite que cada *set* de memória tenha mais blocos.
A vantagem do aumento do grau de associatividade é a redução do nº de colisões, uma vez que com mais blocos de memória, mais endereços podem ser mapeados e assim a probabilidade de se tentar escrever num sítio já ocupado é menor. A diminuição do número de colisões gera por isso um menor número de *misses*, melhorando assim o *hit rate* e diminuindo por consequência o *miss rate*.
A desvantagem deste aumento do grau de associatividade é o aumento do número de endereços possíveis, o que implica que a *tag* usada para mapeamento dos endereços terá que ter mais bits. O tempo de procura e selecção de set pode também aumentar devido à complexidade deste tipo de memória, contribuindo para piores *hit time* e *miss penalty.*
Devido à tecnologia e controlo adicionais associados a um aumento da associatividade, os custos para este tipo de memória são maiores.

Pergunta 5:
A segunda opção é mais amigável da memória. Em C, as matrizes ficam guardadas na memória seguindo uma ordem *row major*, ou seja, primeiro é guardada a 1º linha na totalidade, depois a 2º linha etc…
A função soma2 tira portanto partido da localidade espacial, uma vez que em cada iteração do ciclo acede aos 2 elementos da linha y. Quando o primeiro acesso à linha y da matriz é feito, o outro elemento da linha é também carregado para memória estando "imediatamente" disponível para ser acedido pela segunda atribuição da iteração do ciclo.
A função soma1 como faz o acesso à matriz por colunas, no acesso ao 1º elemento, a 1ª linha da matriz vai ser carregada para memória, no entanto essa linha não vai ser usada por nenhuma operação desse ciclo, assim, na 2ª iteração do ciclo uma nova linha irá ser carregada e na instrução seguinte, será a 1ª linha novamente a ser acedida. Se o valor de N for suficientemente grande é possível que cada acesso a uma nova linha diferente da anterior implique novo carregamento para a memória dessa linha, visto que pode já não estar em cache. Assim sendo, visto que a função no acesso a um novo elemento troca de linha, pode acontecer que todos os acessos à matriz irão gerem *miss*.