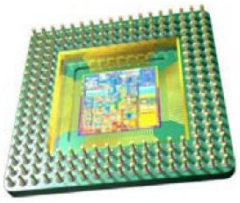
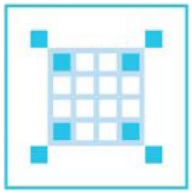


Arquitectura de Computadores

LICENCIATURA EM ENGª INFORMÁTICA
FACULDADE DE CIÊNCIA E TECNOLOGIA
UNIVERSIDADE DE COIMBRA



Lab 10 - Optimização de Código

Neste trabalho laboratorial pretende-se demonstrar como uma programação cuidada pode melhorar dramaticamente a eficiência computacional dos nossos programas. Pretende-se também verificar como a programação em *assembly* pode ajudar a melhorar os tempos de processamento de funções críticas que são executadas múltiplas vezes.

1. Introdução

Nesta ficha vamos recorrer de novo à função de binarização mencionada no trabalho prático 6. O objectivo é perceber como podemos optimizar o código de funções complexas e que são executadas múltiplas vezes. Para isso, na função **`main`** (`main.c`) fornecida com este enunciado, pode verificar (a partir da linha 172) que a função de binarização vai ser chamada 25 vezes, para simular o que eventualmente poderia acontecer caso estivéssemos a processar uma sequência de vídeo em vez de uma única imagem. Pretende-se que a função de binarização possa calcular, numa primeira fase, o valor médio dos pixéis contidos na imagem para depois utilizar esse valor como limiar de binarização. Todos os pixéis abaixo do limiar deverão ser substituídos por 0 e todos os pixéis acima desse limiar deverão ser substituídos por 255. As múltiplas chamadas a esta função de binarização vão ser cronometradas e o valor gasto com a função binariza vai ser impresso no ecrã.

2. Exemplo de Optimização de Código

a) Numa primeira abordagem implemente a função binariza em C (`bin_img_c`). Corra o programa utilizando esta função. Para isso, descomente a linha correspondente à chamada da função no ficheiro `main.c` fornecido com o trabalho. Execute o programa e aponte o tempo de execução que obteve com a utilização desta função.

b) De seguida implemente a função binariza agora escrita directamente em *assembly* do MIPS (`bin_img_asm`). Tal como no ponto anterior, corra agora o programa recorrendo a esta função no ficheiro `main.c`. Anote o tempo de execução obtido e compare com a versão anterior. Qual das versões teve um tempo de execução mais baixo?

c) Os compiladores modernos permitem a activação de modos de optimização de código que permitem gerar programas executáveis mais eficientes. No gcc a flag utilizada para este efeito é a flag -O que pode ainda conter um sufixo numérico que indica qual o

modo de optimização a utilizar. Veja a seguinte página para obter uma explicação dos vários modos de optimização: http://www.rapidtables.com/code/linux/gcc/gcc-o.htm Recorra novamente à versão em C que implementou no ponto a), mas desta feita utilize a flag de optimização na compilação. Execute o programa e verifique os tempos de execução obtidos com cada um dos modos de optimização. Compare estes tempos com o obtido pela sua própria implementação em *assembly*. Caso tenha curiosidade em perceber quais os tipos de optimizações realizadas, compile a função com a opção -S para gerar o código *assembly* correspondente.

d) Procure optimizar o seu código em assembly desenvolvido no ponto b) para ver até que ponto pode diminuir o tempo de execução da sua função. Bata o recorde da sua turma prática, ou pelo menos tente bater os tempos de execução da versão em C com as *flags* de optimização.

NOTA: Para poder controlar manualmente a utilização dos *delayed slots* terá que utilizar a seguinte directiva de compilação (o *assembler* deixa de fazer a reordenação do código e a ocupação automática dos *delayed slots*):

.set noreorder

Note que ao fazer isto terá que preencher manualmente o *delayed slot* após cada instrução de salto (*branches* e *jumps*). Se não conseguir utilizar esse *slot* (por não ser capaz de reordenar o código) deverá utilizar a instrução ***NOP***.

A qualquer momento poderá instruir o assembler para voltar a controlar a reordenação do código fazendo:

.set reorder