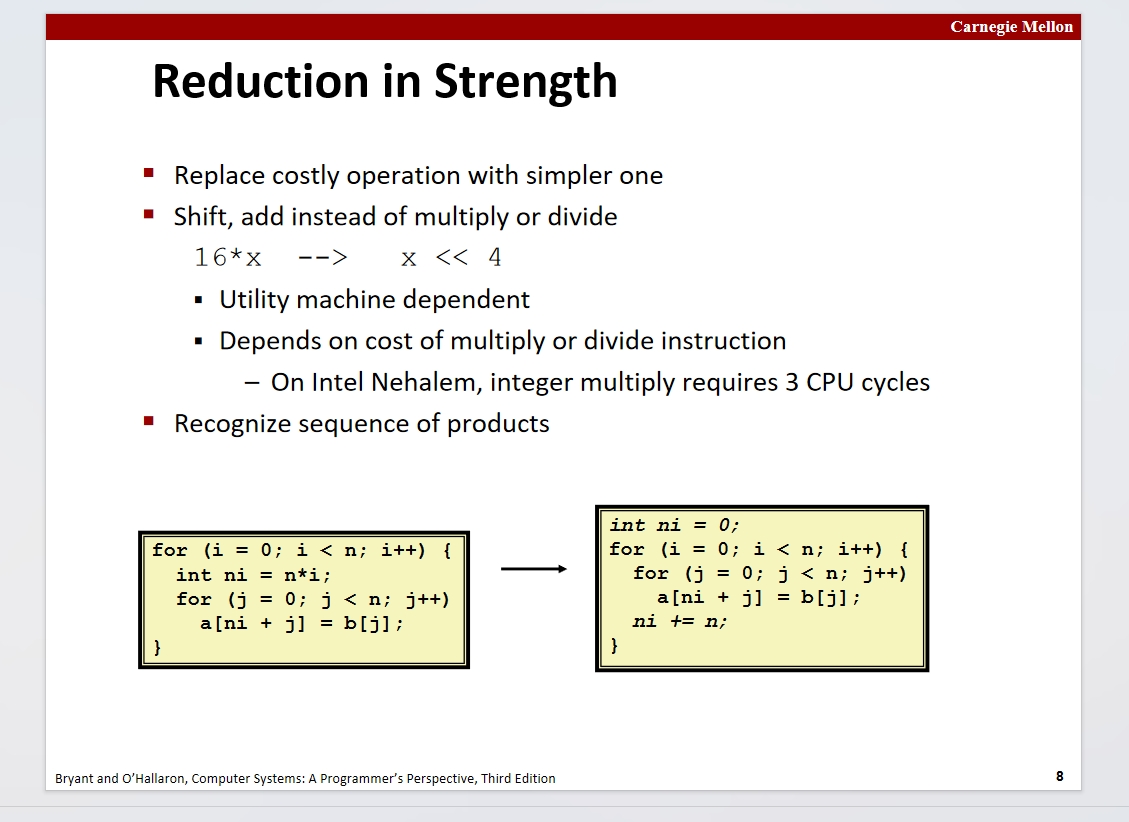


一种优化方式：代码移动。如上，上面的二维矩阵索引中，观察到其实只有j是每次循环都变化的，而n和i都是常量，故这里将n\*i的操作放到循环外面中来，使得只计算一次即可，免去每次循环中的计算，这里即使简单的把部分代码的执行移动位置，使得减少一些操作的次数



还有一种优化方式，由高时钟周期的操作向少时钟周期的操作，就比如乘法操作向加法操作的优化，如上，上面优化前每次外层循环使用乘法操作来计算一个新的操作数，而优化后的操作是在每次外层循环使用一个加法操作来实现，这样多使用了一个外层的变量来替换掉乘法操作来进行优化，而这么做的前提是由于每次的乘法操作都具有一定的规律性，这里即使由于每次的乘法操作都使得数固定的增加一个数值

void lower(char \*s)

{

size\_t i;

for (i = 0; i < strlen(s); i++)

if (s[i] >= 'A' && s[i] <= 'Z')

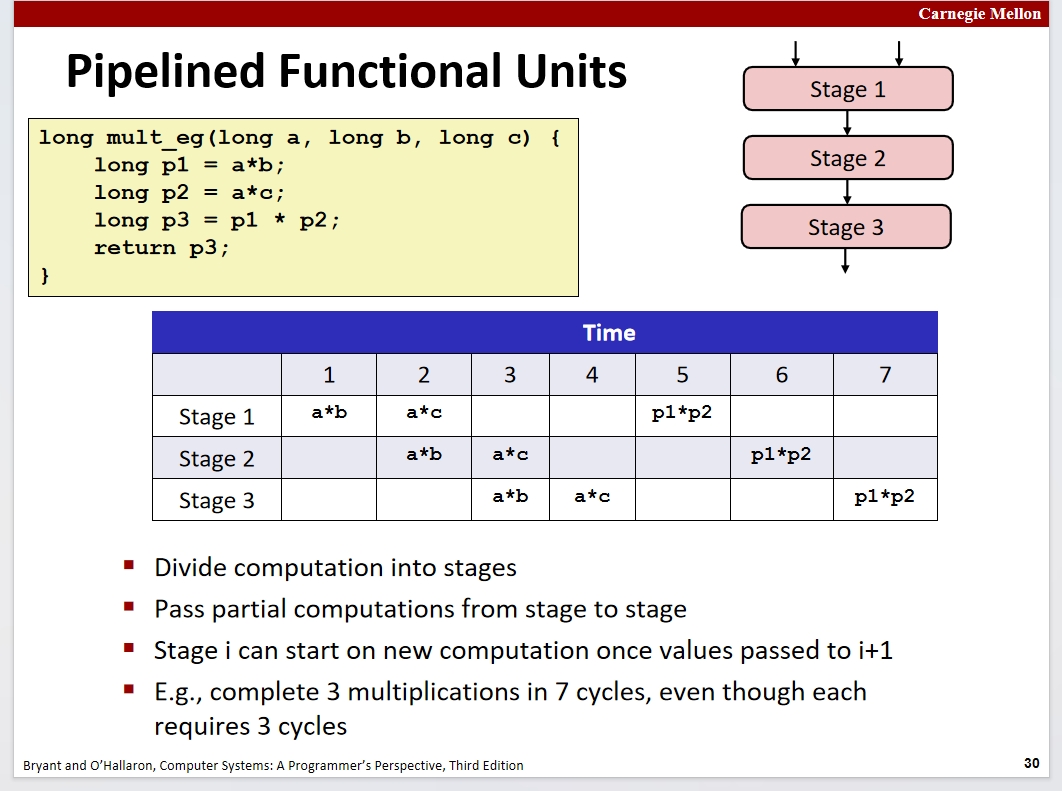
s[i] -= ('A' - 'a');

}

对于这个程序，当输入的s过大时，运行时间会非常的恐怖，原因是由于strlen的复杂度为O(n),而函数位于循环体的条件检测，当每次循环判断时，会运行一次这个函数，故这里整体函数的时间复杂度为O(n2)

优化自然也很简单，就是吧strlen函数定义为一个循环体外的变量直接调用，省去每次循环的函数调用

在编译器优化中，影响优化的主要是别名和函数调用的副作用



这里简单介绍下”流水线”，需要注意，这里的a\*b和a\*c操作不是相关联的，是可以独立运行的操作，故这里将它俩用一定顺序来进行执行的安排，这里安排将a\*c放在a\*b后面执行，即当a\*b执行完后立即执行a\*c，这里之所以有三个是因为编译器通常不能一步就执行完一个操作，大部分情况都是将一个操作分为几个小操作来进行实现的，这里也是如此，当进入第二个时钟周期时，程序会执行stage1的a\*c，同时，由于一个操作间互相独立，故在此时也会执行先前a\*b操作的下一步，以此类推