Mikhail Skobov  
CSEP 590 – Programming Systems  
Homework 2

**Problem 1:**

Note: not entirely familiar with x86 assembly, so adding comments to specify intent. Also, compiler is not going to create meaningful labels on your behalf, those are there for clarity.

BEGIN:

mov eax, [ebp-4] ; Move x to eax  
 mov edx, [ebp-8] ; Move y to edx  
 cmp edx, 0 ; Compare y to 0  
 jng LOOP ; Jump over the if statement if not-greater  
 inc eax ; Increment x by 1  
LOOP:  
 cmp eax, 0 ; Compare x to 0  
 jng LOOP\_END ; Leave loop if x not GT 0  
 dec eax ; Decrement x  
 j LOOP ; Unconditional jump back to loop  
LOOP\_END:  
 mov [ebp-4], eax ; Store x (may not be necessary)

NOTE: If the optimizer was smart enough, it may also remove the while loop altogether, and just move 0 to x (mov [ebp-4], 0)

**Problem 2:**

In the paper, Briggs, Cooper, and Torczon set out to describe the improvements they made to Chaitin’s graph-coloring register allocator. They begin by elaborating on register allocation, graph-coloring, Chaitin’s allocator, and finally stating several deficiencies of Chaitin’s approach, with two concrete examples.

One of the first problems mentioned relates to how Chaitin’s allocator selects the spilling candidate; according to the authors, Chaitin’s allocator chooses a spill candidate based on the ratio of spill cost (the run-time cost of additional load/store instructions) to the current degree of the node. According to the authors, this criteria is not strong enough, since it would not be able to find a k-coloring even if one exists, due to the allocator’s “pessimism”. They show how this might happen using the “diamond” example, where a reader can see that the graph can be 2-colored, but the allocator cannot produce that result (although they do state that in practice, the allocator generally performs well).

The second deficiency claimed has to do with rematerialization, or re-computing values in a single instruction and placing them into the desired register – an example of which would be loading a constant into a register. Chaitin’s allocator is able to correctly handle rematerialization with a single value in a live range, but is not able to handle more complex cases.

The authors proposed two solutions to address the problems above: optimistic coloring and extending rematerialization for complex, multi-valued live ranges. For optimistic coloring, they wanted to address the way Chaitin’s allocator handled spilling, and effectively solve the “diamond” and “SVD” problems. In order to achieve this, they combined Matula and Beck’s stronger coloring, with Chaitin’s mechanism for cost-guided spill selection. To do this, they modified the “simplify” stage of the allocator to “optimistically” push spill candidates onto the stack, hoping that a color will become available (disregarding the high degree of the candidate), and in the “select” stage, their algorithm attempts to color remaining nodes, and if unsuccessful, it inserts spill code, rebuilds the interference graph, and tries again. This deferral of the spill decision has 2 powerful side effects: it eliminates non-productive spills, and creates a more powerful coloring heuristic, which is capable of finding a k-coloring for k-colorable graphs.

For rematerialization, the authors took a new approach: their allocator split each live range into its component values, tagged each value with rematerialization information, and formed new live ranges from connected values having identical tags. This new approach, however, introduced an issue with unnecessary splits, which the authors attempted to solve via conservative coalescing and biased coloring. Conservative coalescing essentially added a constraint on coalesce that states: only combine two live ranges if the resulting live range will not be spilled. Biased coloring uses the select stage of the allocator to try and assign the same color to two live ranges that are connected by a copy. When used together, these two mechanisms remove most unproductive copies.

When viewing the empirical results of the experiment, it is evident that the authors succeeded in their endeavors of improving Chaitin’s allocator. The improved allocator was able to reduce the cycles of spill code by a lot with both optimistic coloring and improved rematerialization, at the cost of a longer compile time.

Discussion topic/question:

* Why did they attempt to do everything automatically, rather than taking a top-down approach and passing valuable variable/register information through the IR (such as "loop conditional" flag)?
* Soft topic: does anyone else think they did a poor job explaining live ranges in the paper? This is a fairly esoteric topic, so it makes it tough to analyze without a more detailed explanation/examples.