# CS143
# Compilers

Lecture 12
August 13, 2001

## Next-Use Information

- To allocate registers efficiently, it is helpful to determine where a name in a three-address statement is used once its value has been assigned.
- Once a name's value is no longer needed, if it is stored in a register, the register can be assigned to another name.
- Furthermore, after the value is no longer needed, the temporary variable holding that value can be re-used.



## Computing Next Uses

- Next-use information can be determined by scanning backwards through a basic block.
- Initially, all non-temporary variables are considered live upon exit from the block. If temporaries can be re-used across blocks, they must be considered live as well.
- When a statement x := y op z is encountered, x is marked as "not live" and "no next use" for preceding statements, and the next use of y and z is set to this statement. This information is recorded in the symbol table.



## Storage for Temporary Names

- Next-use information can be very helpful in determining when temporary variables can be re-used.
- Specifically, the lifetime of temporary variables can be determined by computing next uses. When a temporary $t_1$ ceases to be live, it can be reused, thus replacing another temporary variable $t_2$ whose lifetime does not overlap with that of $t_1$.



## A Simple Code Generator

- We now outline a simple code generator that translates three-address statements into target code, using registers whenever possible.
- We make the following assumptions:
  - Operators in the source language correspond to instructions in the target language
  - Values in registers can be left in registers as long as possible, storing them only at the end of a basic block. This requirement can be relaxed with extra effort, but is used here for simplicity.



## Register and Address Descriptors

- To use registers as efficiently as possible, we make use of a register descriptor, which keeps track of which name has its value stored in each register.
- An address descriptor indicates where the current value of a given name is stored, whether it is in memory, a register, or a stack location.
- These descriptors facilitate the assignment of registers to variables during translation.

## The *getreg* Function

- Our code-generator relies on a function getreg, which returns a location L that will hold the value of x for the assignment x := y op z. A simple implementation proceeds as follows:
  - If y's register is no longer need by y, use it
  - Otherwise, select an empty register
  - If no empty register is available, and x is used again in this basic block, select a register R and move its data to memory
  - If all else fails, use the memory location for x.



## A Code-Generation Algorithm

Given the preceding foundation, our code generator proceeds as follows, to translate each statement of the form x := y op z within a basic block:

- Use getreg to determine the location L that will hold the value of x
- Use the address descriptor to determine the most accessible location of the value of y. Then, output MOV y, L.
- Output the target statement op z,L. Update the address descriptor for x and the register descriptor of L, if applicable.
- If y and z have no next uses, update the register descriptor accordingly.



## Conditional Statements

- There are two common approaches to generating code for conditional statements:
  - Branch if the value of a given register is either: negative, zero, positive, nonnegative, nonzero, or nonpositive. This approach requires a subtraction.
  - Maintain a set of condition codes to indicate whether the last computational result is positive, negative, or zero. With this approach, which doesn't require any arithmetic, a condition-code descriptor can keep track of results of conditional statements.



## Register Allocation and Assignment

- One approach is to assign different types of values (e.g. base addresses, arithmetic computations, etc.) to different registers. While simple, it can be wasteful.
- In global register allocation, registers are assigned to frequently-used variables, eliminating storage at the end of each basic block for such variables.
- If a variable is used within a loop, the savings arising from storing that variable in a register during the loop can be approximated based on its usage outside of the loop, to determine if such a register should be used.
- It must also be determined whether outer loop variables should be stored in registers during inner loops, which is expensive but may also be worthwhile.



## Graph Coloring

- When registers are scarce, data from registers must be written to memory to ensure availability.
- One systematic technique for efficient allocation begins by assigning registers as if infinitely many were available. These symbolic registers essentially correspond to names in the intermediate code.
- Then, a register-interference graph is constructed, having these virtual registers as nodes and two nodes are connected if one is live at a point where the other is defined.
- The nodes of the graph are colored so that adjacent nodes have different colors. Each color corresponds to a unique register.