# Code Generation – Part 3

Y. N. Srikant
Computer Science and Automation
Indian Institute of Science
Bangalore 560 012

NPTEL Course on Compiler Design

# Outline of the Lecture

1. Code generation – main issues
2. Samples of generated code
3. Two Simple code generators
4. Optimal code generation
   a) Sethi-Ullman algorithm
   b) Dynamic programming based algorithm
   c) Tree pattern matching based algorithm
5. Code generation from DAGs
6. Peephole optimizations

Topics 1,2,3,4, and 5 were covered in parts1 and 2 of the lecture.

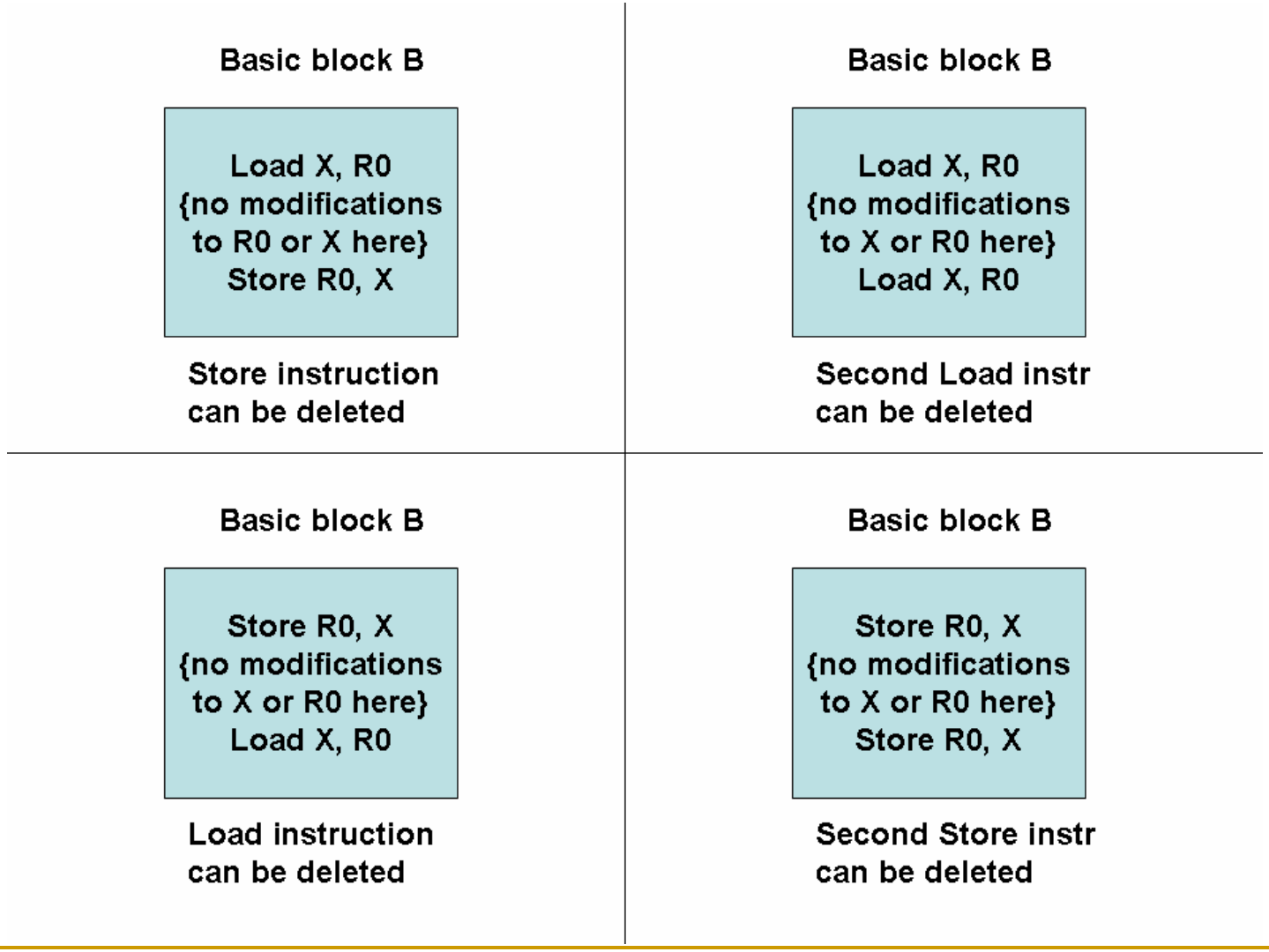
# Peephole Optimizations

- Simple but effective local optimization
- Usually carried out on machine code, but intermediate code can also benefit from it
- Examines a sliding window of code (peephole), and replaces it by a shorter or faster sequence, if possible
- Each improvement provides opportunities for additional improvements
- Therefore, repeated passes over code are needed

# Peephole Optimizations

- Some well known peephole optimizations
  - eliminating redundant instructions
  - eliminating unreachable code
  - eliminating jumps over jumps
  - algebraic simplifications
  - strength reduction
  - use of machine idioms

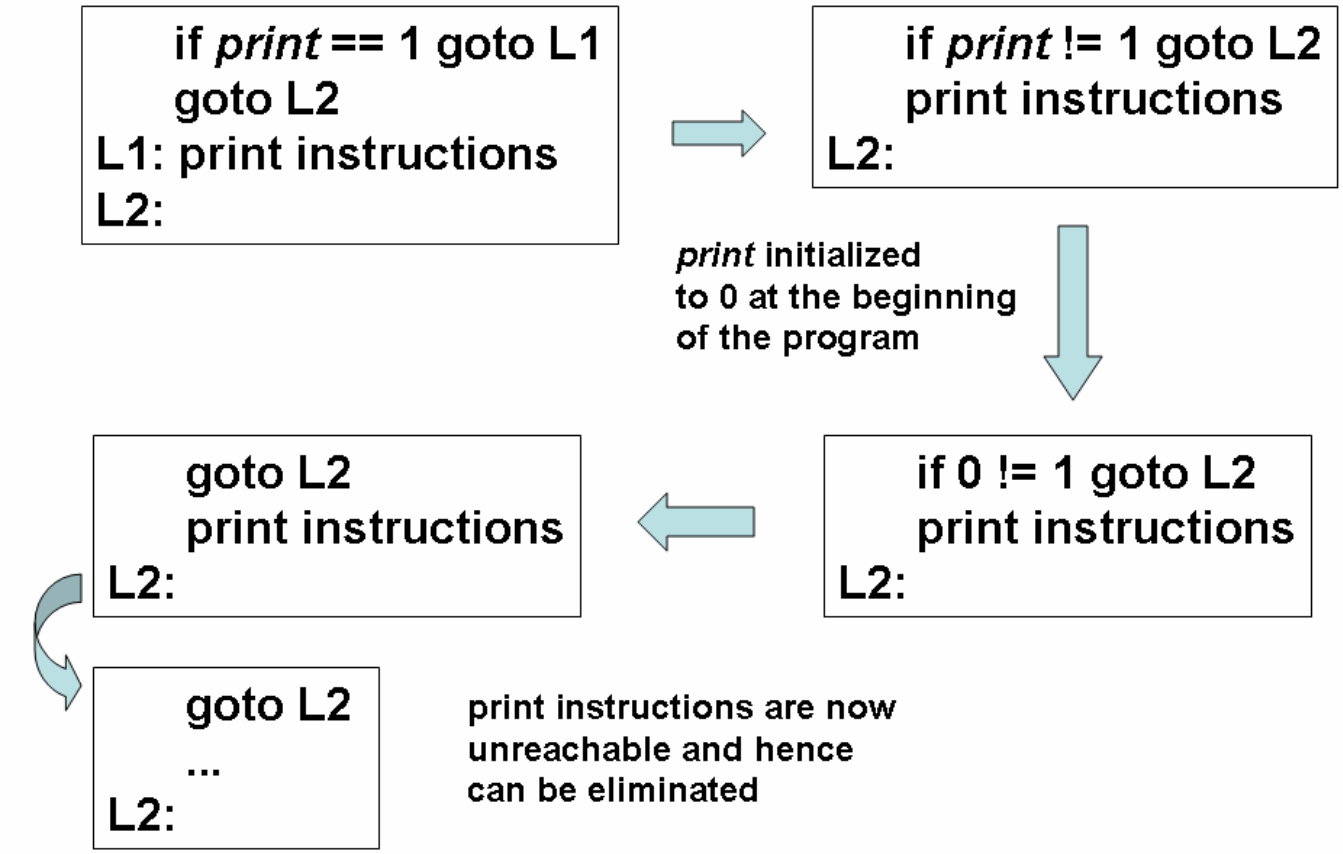
# Elimination of Redundant Loads and Stores



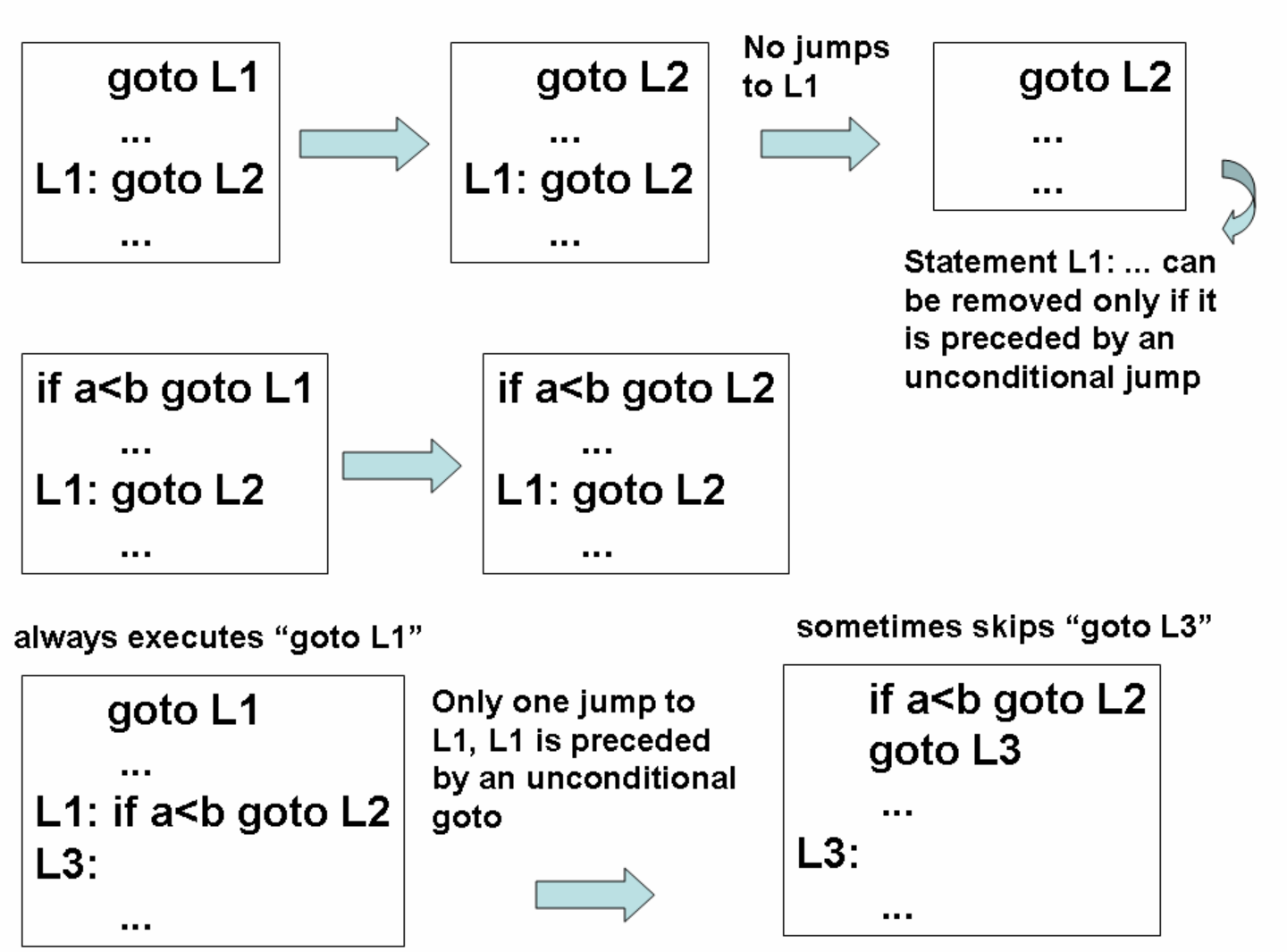
# Eliminating Unreachable Code

- An unlabeled instruction immediately following an unconditional jump may be removed
  - May be produced due to debugging code introduced during development
  - Or due to updates to programs (changes for fixing bugs) without considering the whole program segment

# Eliminating Unreachable Code



# Flow-of-Control Optimizations



# Reduction in Strength and Use of Machine Idioms

- $x^2$ is cheaper to implement as x*x, than as a call to an exponentiation routine
- For integers, $x*2^3$ is cheaper to implement as x << 3 (x left-shifted by 3 bits)
- For integers, $x/2^2$ is cheaper to implement as x >> 2 (x right-shifted by 2 bits)

# Reduction in Strength and Use of Machine Idioms

- Floating point division by a constant can be approximated as multiplication by a constant
- Auto-increment and auto-decrement addressing modes can be used wherever possible
  - Subsume INCREMENT and DECREMENT operations (respectively)
- Multiply and add is a more complicated pattern to detect

# Code Generation: State-of-the-Art and Future Directions

- *gnu* provides a code generator generator
  - Takes machine description in *register transfer language (rtl)*
  - Incorporates several optimizations (peephole, instruction scheduling, register allocation etc.)
  - Generates efficient code generators
  - Tedious to use – *rtl descriptions are hard to understand and write!*
  - Not easy to retarget to special processors, such as DSP.

# Code Generation: State-of-the-Art and Future Directions

- Tree pattern matching based CGGs are becoming popular
  - No commercial packages available today
- Combining instruction selection and scheduling is still not possible
- Instruction selection with *power consumed* as the criterion is still not possible
  - requires power consumption information from the chip manufacturer, and
  - facilities on the chip to turn off/on functional units/memory banks etc., and
  - energy profiling of programs to identify ‘hot/idle’ regions