



中山大學  
SUN YAT-SEN UNIVERSITY



国家超级计算广州中心  
NATIONAL SUPERCOMPUTER CENTER IN GUANGZHOU

# Compilation Principle 编译原理

---

## 第22讲：代码优化(2)

张献伟

[xianweiz.github.io](https://xianweiz.github.io)

DCS290, 06/17/2021



中山大學  
SUN YAT-SEN UNIVERSITY



# Review Questions

- Q1: Code optimizations are to generate better code, list some 'better' metrics.

Execution time, memory usage, energy, binary size...

- Q2: what is a Basic Block?

A straight-line sequence of code with only one entry point and only one exit.

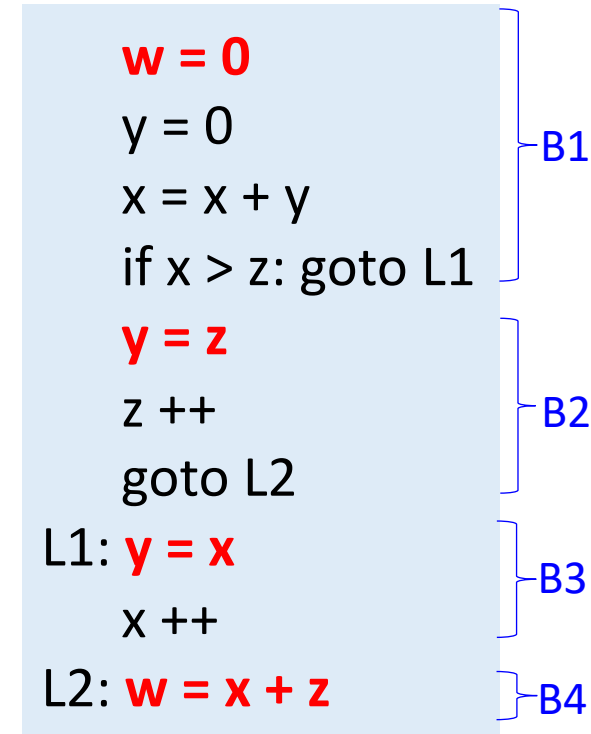
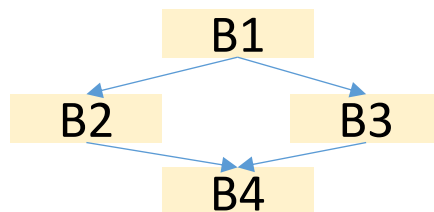
- Q3: how to partition code into BBs?

Identify leader insts; a BB consists of a leader inst and subsequent insts before next leader.

- Q4: What is a control-flow graph?

A directed graph where nodes are BBs, edges show flow of execution between BBs.

- Q5: What is the CFG of the listed code?



# Local and Global Optimizations

---

- Local optimizations[局部优化]
  - Optimizations performed exclusively within a basic block
  - Typically the easiest, never consider any control flow info
    - All instructions in scope executed exactly once
  - Examples:
    - constant folding[常量折叠]
    - common subexpression elimination[删除公共子表达式]
- Global optimizations[全局优化]
  - Optimizations performed across basic blocks
    - Scope can contain if / while / for statements
    - Some insts may not execute, or even execute multiple times
  - Note: global here doesn't mean across the entire program
    - We usually optimize one function at a time

# ## Local Optimization: Examples

- Common subexpression elimination[删除公共子表达式]
  - Two operations are common if they produce the same result
    - It is likely more efficient to compute the result once and reference it the second time rather than re-evaluate it[避免重复计算]
- Dead code elimination[删除无用代码]
  - If an instruction's result is never used, the instruction is considered “dead” and can be removed from the instruction stream[结果从不使用]

```
y = x + z;  
y = x * x + (x/y)  
z = x * x + y;
```



```
y = x + z;  
t1 = x * x  
t2 = x / y  
y = t1 + t2  
t3 = x * x  
z = t3 + y;
```



```
y = x + z;  
t1 = x * x  
t2 = x / y  
y = t1 + t2  
t3 = x * x  
z = t1 + y;
```

# DAG of Basic Blocks

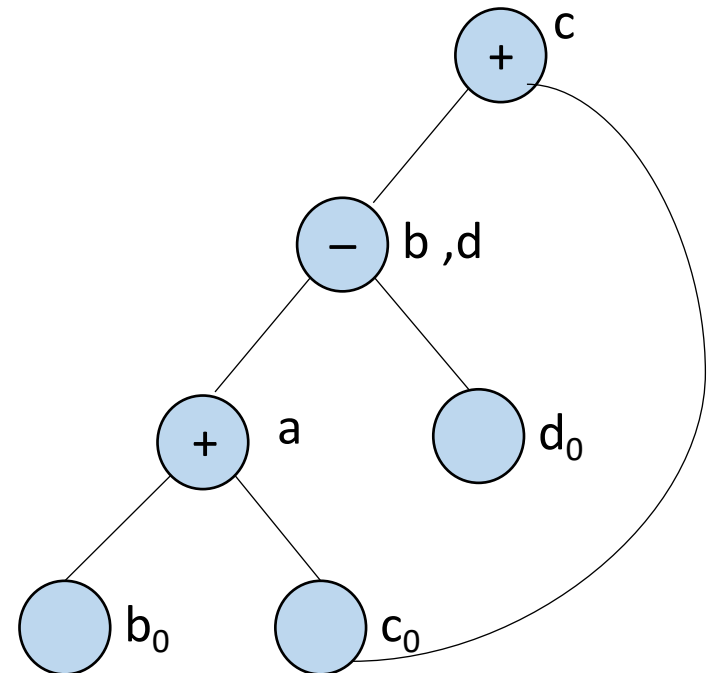
---

- Many important techniques for local optimization begin by transforming a BB into a DAG (directed acyclic graph)[无环有向图]
- To construct a DAG for a BB as follows
  - Create a node for each of the initial values of the variables appearing in the BB[为变量初始值创建节点，叶子]
  - Create a node  $N$  associated with each statement  $s$  within the block[为声明语句创建节点，中间]
    - The children of  $N$  are those nodes corresponding to statements that are the last definitions, prior to  $s$ , of the operands used by  $s$
    - Label  $N$  by the operator applied at  $s$ [用运算符标注节点]
  - Certain nodes are designated output nodes[某些为输出节点]
    - These are the nodes whose variables are live on exit from the block (i.e., their values may be used later, in another block of the flow graph)

# Example: DAG

- (3)  $c = b + c$ 
  - $b$  refers to the node labelled '-'
    - Most recent definition of  $b$
- (4)  $d = a - d$ 
  - Operator and children are the same as the 2<sup>nd</sup> statement
    - Reuse the node

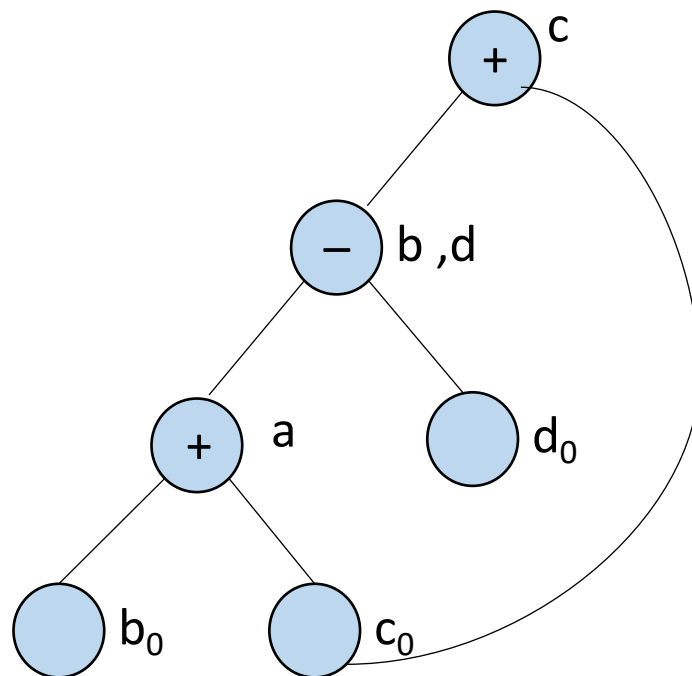
(1)  $a = b + c$   
(2)  $b = a - d$   
(3)  $c = b + c$   
(4)  $d = a - d$



# Local Opt.: Elimination

- If  $b$  is **not live** on exit from the block
  - No need to keep  $b = a - d$
- If both  $b$  and  $d$  are **live**
  - Remove either (2) or (4) :  
**common subexpr elimination**
  - Add a 4<sup>th</sup> statement to copy one to the other
- If only  $a$  is **live** on exit
  - Then remove nodes from the DAG correspond to dead code
    - $c \rightarrow b, d \rightarrow d_0$
  - This is actually **dead code elimination**

(1)  $a = b + c$   
(2)  $b = a - d$   
(3)  $c = b + c$   
(4)  $d = a - d$

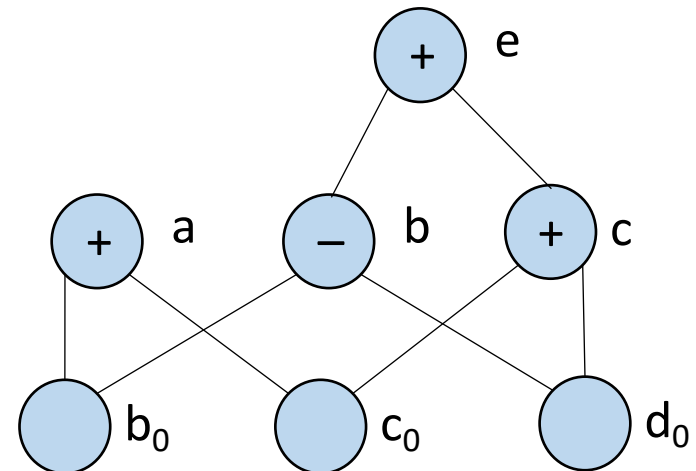


# Local Opt.: Elimination (cont.)

- When finding common subexprs, we really are finding exprs that are guaranteed to compute the same value, no matter how that value is computed[过于严苛]
  - Thus miss the fact that (1) and (4) are the same
    - $b + c = (b - d) + (c + d) = b_0 + c_0$

(1)  $a = b + c$   
(2)  $b = b - d$   
(3)  $c = c + d$   
(4)  $e = b + c$

- **Solution:** algebraic identities[代数恒等式]





# Local Opt.: Algebraic Identities[代数恒等式]

- Eliminate computations by applying mathematical rules[使用数学规则]
  - Identities:  $a * 1 \equiv a$ ,  $a * 0 \equiv 0$ ,  $b \& \text{true} \equiv b$
  - Reassociation and commutativity[重组、交换]
    - ▣  $(a + b) + c \equiv a + (b + c)$ ,  $a + b \equiv b + a$
- **Strength Reduction**[强度削减]
  - Replacing expensive operations (*multiplication, division*) by less expensive operations (*add, sub, shift*)
  - Some ops can be replaced with cheaper ops
  - Examples
    - ▣  $x = y / 8 \rightarrow x = y \gg 3$
    - ▣  $y = y * 8 \rightarrow y = y \ll 3$
    - ▣  $x^2 \rightarrow x * x$
    - ▣  $2 * x \rightarrow x + x$

# Local Opt.: Constant Folding[常量折叠]

---

- **Constant Folding**

- Computing operations on constants at compile time
- Example:

```
#define LEN 100  
x = 2 * LEN;  
if (LEN < 0) print("error");
```

- After constant folding

```
x = 200;  
if (false) print("error");
```

- Dead code elimination can further remove the above *if* statement
- Inherently local since scope limited to statement

# Local Opt.: Constant Propagation[常量传播]

- **Constant Propagation**

- Substituting values of known constants at compile time
- Local Constant Propagation (LCP)

```
x = 3;  
y = x * 2;
```



```
x = 3;  
y = 3 * 2;
```



```
x = 3;  
y = 6;
```

- Some optimizations have both local and global versions
  - Global Constant Propagation (GCP)

```
a = 1;  
x = 3;  
if (...)  
    x = a + 2;  
y = x;
```



```
a = 1;  
x = 3;  
if (...)  
    x = 1 + 2;  
y = x;
```

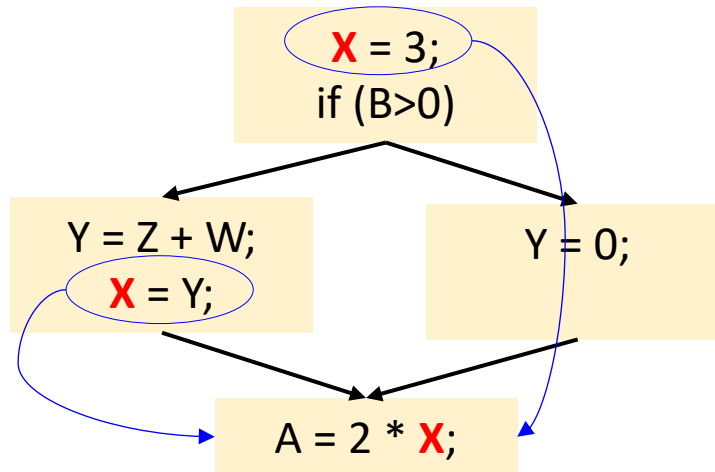
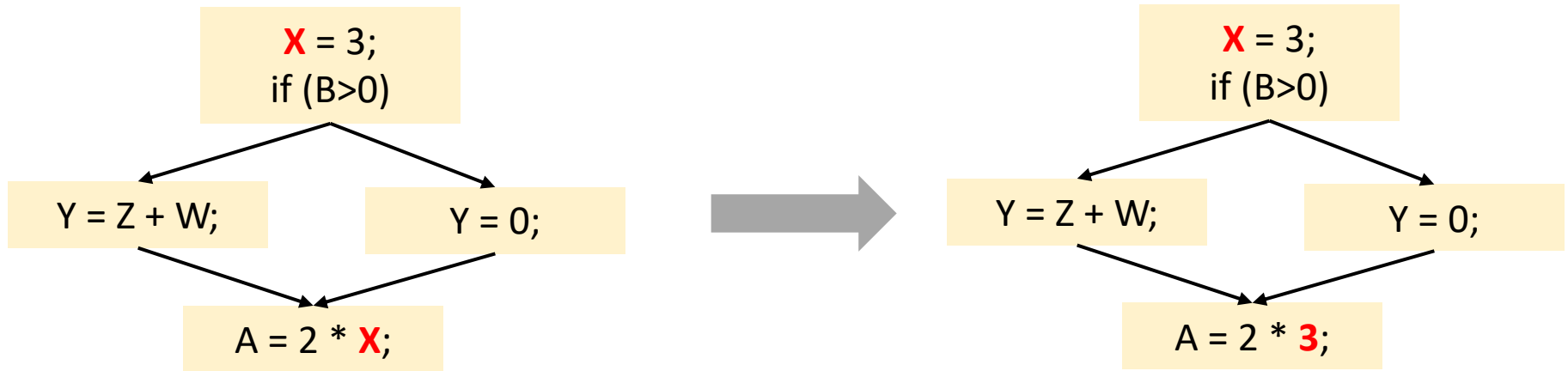


```
a = 1;  
x = 3;  
if (...)  
    x = 3;  
y = 3;
```

- GCP more powerful than LCP but also more complicated
  - Must determine x is constant across all paths reaching x

# ## Global Optimizations

- Extend optimizations to flow of control, i.e. CFG
  - Along **all paths**, the last assignment to X is “X=C”
  - Optimization must be stopped if incorrect in even one path



# Global Opt.: Conservative[需保守]

---

- Compiler must prove some property X at a particular point
  - Need to prove at that point property X holds along all paths
  - Need to be **conservative** to ensure correctness
    - An optimization is enabled only when X is definitely true
    - If not sure if it is true or not, it is safe to say **don't know**
    - If analysis result is **don't know**, no optimization done
    - May lose opt. opportunities but guarantees correctness
- Property X often involves data flow of program
  - E.g. Global Constant Propagation (GCP):  
`X = 7;`  
...  
`Y = X + 3;` // Does value of 7 flow into this use of X?
  - Needs knowledge of **data flow**, as well as control flow
    - Whether data flow is interrupted between points A and B

# Global Opt.: Data Flow[数据流]

---

- Most optimizations rely on a property at given point
  - For Global Constant Propagation (GCP):  
 $A = B + C$ ; // Property:  $\{A=?, B=10, C=?\}$
  - After optimization:  
 $A = 10 + C$ ;
- For this discussion, let's call these properties *values*
- **Dataflow analysis**: compiler analysis that calculates values for each point in a program
  - Values get propagated from one statement to the next
  - Statements can modify values (for GCP, assigning to vars)
  - Requires CFG since values flow through control flow edges
- **Dataflow analysis framework**: a framework for dataflow analysis that guarantees correctness for **all paths**
  - Does *not* traverse all possible paths (could be infinite)
  - To be feasible, makes **conservative** approximations

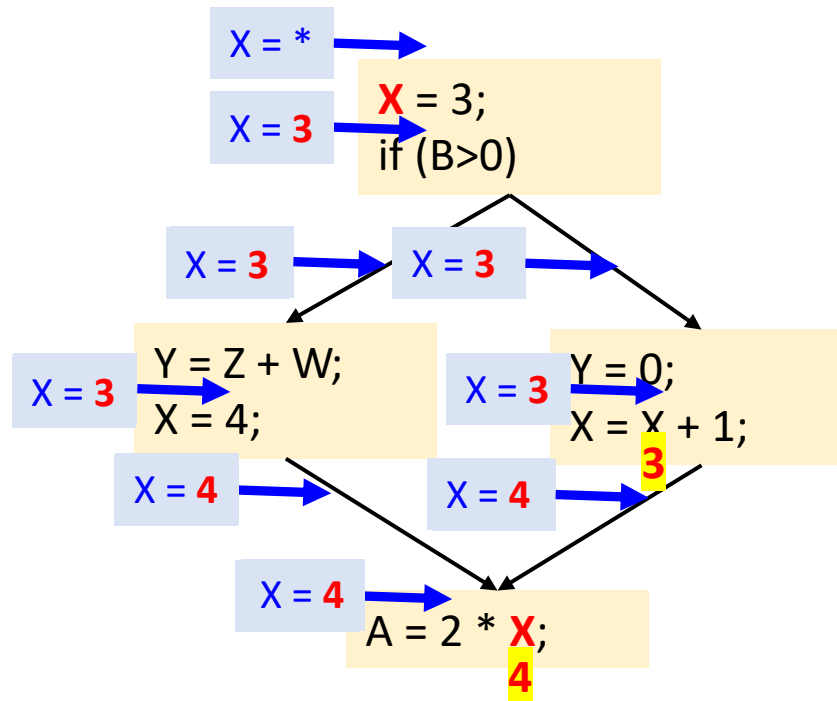
# Global Constant Propagation (GCP)

---

- Let's apply dataflow analysis to compute values for GCP
  - Emulates what human does when tracing through code
- Let's use following notation to express the state of a var:
  - $x=*$ : not assigned (default)
  - $x=1, x=2, \dots$ : assigned to a constant value
  - $x=\#$ : assigned to multiple values
- All values start as  $x=*$  and are iteratively refined
  - Until they stabilize and reach a fixed point
- Once fixed point is reached, can replace with constants:
  - $x=*$ : replace with any constant (typically 0)
  - $x=1, x=2, \dots$ : replace with given constant value
  - $x=\#$ : cannot do anything

# Example

- In this example, constants can be propagated to  $X+1$ ,  $2*X$
- Statements visited in reverse postorder (predecessor first)



$x = *$ : not assigned (default)

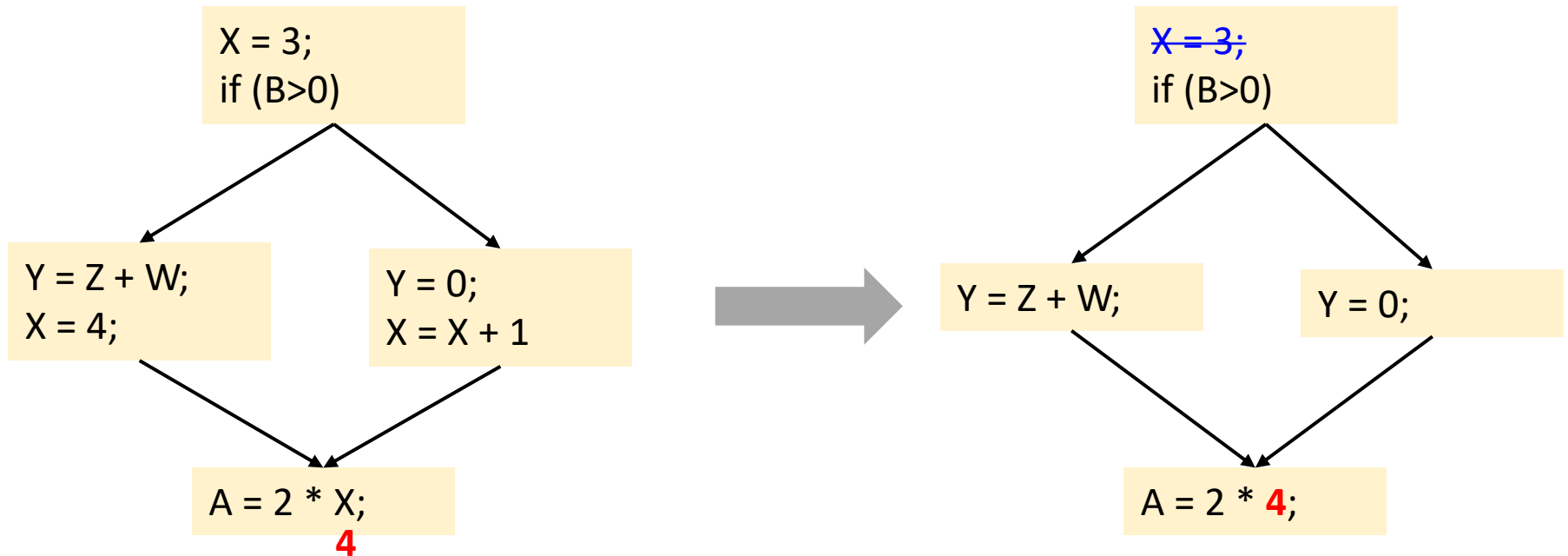
$x = 1, x = 2, \dots$ : assigned to a constant value

$x = \#$ : assigned to multiple values



# Example (cont.)

- Once constants have been globally propagated, we would like to eliminate the dead code



# Machine Optimizations[机器相关优化]

---

- After performing IR optimizations
  - We need to further convert the optimized IR into the target language (e.g. assembly, machine code)
- Specific machines features are taken into account to produce code optimized for the particular architecture[考虑特定的架构特性]
  - E.g., specialized instructions, hardware pipeline abilities, register details
- Typical machine optimizations[典型的优化方案]
  - **Instruction selection and scheduling**: select insts to implement the operators in IR
  - **Register allocation**: map values to registers and manage
  - **Peephole optimization**: locally improve the target code

# Instruction Selection[指令选取]

- To find an efficient mapping from the IR of a program to a target-specific assembly listing[IR到汇编的映射]
- Instruction selection is particularly important when targeting architectures with CISC (e.g., x86)
  - In these architectures there are typically several possible implementations of the same IR operation, each with different properties
  - e.g., on x86 an addition of one can be implemented by an *inc*, *add*, or *lea* instruction

$x = y + z$

```
MOV y,R0  
ADD z,R0  
MOV R0,x
```

$a = a + 1$

```
MOV a,R0  
ADD #1,R0  
MOV R0,a
```



```
MOV a,R0  
INC R0  
MOV R0,a
```

# Instruction Scheduling[指令调度]

- Some facts
  - Instructions take clock cycles to execute (latency)
  - Modern machines issue several operations per cycle (Out-of-Order execution)
  - Cannot use results until ready, can do something else
  - Execution time is order-dependent
- Goal: reorder the operations to minimize execution time
  - Minimize wasted cycles
  - Avoid spilling registers
  - Improve locality

```
A = x * y;  
B = A + 1;  
C = y;
```



```
A = x * y;  
C = y;  
B = A + 1;
```

(Now C=y; can execute while waiting for A=x\*y;)

# Register Allocation[寄存器分配]

---

- In TAC, there are an unlimited number of variables
  - On a physical machine there are a small number of registers
- **Register allocation** is the process of assigning variables to registers and managing data transfer in and out of registers
  - How to assign variables to finitely many registers?
  - What to do when it can't be done?
  - How to do so efficiently?
- Using registers intelligently is a critical step in any compiler
  - Accesses to memory are costly, even with caches
  - A good register allocator can generate code orders of magnitude better than a bad register allocator

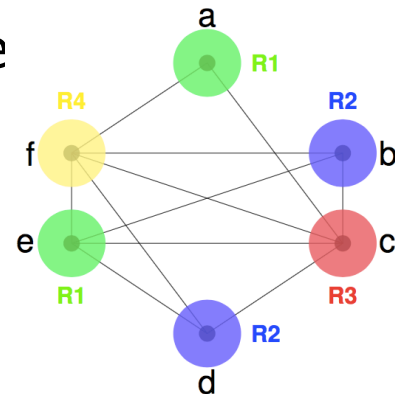
# Register Allocation (cont.)

---

- Goals of register allocation
  - Keep frequently accessed variables in registers
  - Keep variables in registers only as long as they are live
- Local register allocation[局部]
  - Allocate registers basic block by basic block
  - Makes decisions on a per-block basis (hence ‘local’)
- Global register allocation[全局]
  - Makes global decisions about register allocation such that
    - Var to reg mappings remain consistent across blocks
    - Structure of CFG is taken into account on decisions
- Three well-known register allocation algorithms
  - Graph coloring allocator[图着色]
  - Linear scan allocator[线性扫描]
  - LP (Integer Linear Programming) allocator[整数线性规划]

# Graph Coloring[图着色]

- Register interference graph (RIG)[相交图]
  - Each node represents a variable
  - An edge between two nodes  $V_1$  and  $V_2$  represents an interference in live ranges[活跃期/生存期]
- Based on RIG,
  - Two variables can be allocated in the same register if there is no edge between them[若无边相连, 可使用同一寄存器]
  - Otherwise, they cannot be allocated in the same register
- Problem of register allocation maps to graph coloring
  - Once solved,  $k$  colors can be mapped back to  $k$  registers
  - If the graph is  $k$ -colorable, it's  $k$ -register-allocatable



# Register Spilling[寄存器溢出]

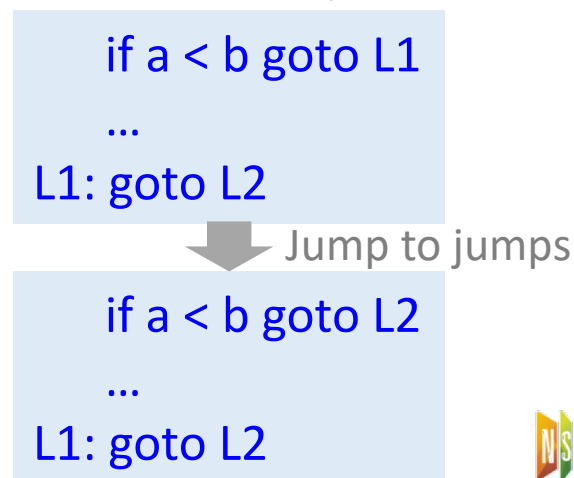
---

- Determining whether a graph is  $k$ -colorable is NP-complete
  - Therefore, problem of  $k$ -register allocation is NP-complete
  - In practice: use heuristic polynomial algorithm that gives close to optimal allocations most of the time
  - Chaitin's graph coloring is a popular heuristic algorithm
    - E.g. most backends of GCC use Chaitin's algorithm
- What if  $k$ -register allocation does not exist?
  - Spill a variable to memory to reduce RIG and try again
  - Spilled var stays in memory and is not allocated a reg
- Spilling is slow
  - Placed into memory, loaded into register when needed, and written back to memory when no longer used



# Peephole Optimization[窥孔优化]

- Optimization ways
  - Usual: produce good code through careful inst selection and register allocation
  - Alternative: generate naïve target code and then improve
- A simple but effective technique for locally improving the target code[很局部的优化，但可能带来性能的极大提升]
  - Done by examining a sliding window of target instructions (called **peephole**) and replacing instruction sequences within the peephole by a shorter or faster sequence, whenever psbl
  - Can also be applied directly after IR generation to improve IR
- Example transformations
  - Redundant-instruction elimination
  - Flow-of-control optimizations
  - Algebraic simplifications
  - Use of machine idioms



# Summary[总结]

---

- Code can be optimized at different levels with various techniques
  - IR: local, global, common subexpression elimination, constant folding and propagation, ...
  - Target: instruction, register, peephole, ...
- Interactions between the various optimization techniques
  - Some transformations may expose possibilities for others
  - One opt. may obscure or remove possibilities for others
- Affect of compiler opts are intertwined and hard to separate
  - Finding optimal opt combinations is in itself research
  - Compilers package opts that typically go together into levels (e.g -O1, -O2, -O3)



The **END** is Near

# 期末考试

- 编译原理

- 课堂参与（10%）- 点名、提问、测试
- 课程作业（20%）- 4次左右，理论
- 期中考查（10%）- 课下习题
- 期末考试（60%）- 闭卷

[3次测试]

[2次作业]

[1次练习]

参考答案已上传

- 考试时间

- 14:30-16:30
- 7月8日(周四)

- 题目类型

- 判断题
- 简答题
- 应用题
- 综合应用题

- 主要内容

- 词法分析
- 语法分析
- 语义分析
- 代码生成及优化

# Exam should NOT be the END ...

Front-end  
(15x2学时)

Back-end  
(7x2学时)

Talk  
(2x2学时)

Intro+Review  
(2x2学时)

期末考试分值分布

科研/工作相关性

