CODE GENERATION - II

Unit - IV

Design of Code Generator

Overview

- Transformations on Basic blocks
- Code Generation Algorithm
- Peephole Optimization

Equivalence of Basic Blocks

- Two basic blocks are (semantically) *equivalent* if they compute the same set of expressions

```
b  := 0
t1 := a + b
t2 := c * t1
a  := t2
```

↓

```
a := c*a
b := 0
```

```
a  := c * a
b  := 0
```

↓

```
a := c*a
b := 0
```

Blocks are equivalent, assuming `t1` and `t2` are *dead*: no longer used (no longer *live*)

Transformations on Basic Blocks

- A code-improving transformation is a code optimization to improve speed or reduce code size
- Global transformations are performed across basic blocks
- Local transformations are only performed on single basic blocks
- Transformations must be safe and preserve the meaning of the code
- Number of transformations can be applied to a basic block without changing the set of expressions computed by the block.
- Two important classes of transformation are :
  - Structure-preserving transformations
    1. common subexpression elimination
    2. dead-code elimination
    3. renaming of temporary variables
    4. interchange of two independent adjacent statements
  - Algebraic transformations

Common-Subexpression Elimination

- Remove redundant computations

```
a := b + c
b := a - d
c := b + c
d := a - d
```

```
a := b + c
b := a - d
c := b + c
d := b
```

```
t1 := b * c
t2 := a - t1
t3 := b * c
t4 := t2 + t3
```

```
t1 := b * c
t2 := a - t1
t4 := t2 + t1
```

Dead Code Elimination

- Remove unused statements - Suppose x is dead, that is, never subsequently used, at the point where the statement x : = y + z appears in a basic block. Then this statement may be safely removed without changing the value of the basic block.

```
b := a + 1
a := b + c
…
```

```
b := a + 1
…
```

Assuming **a** is *dead* (not used)

```
if true goto L2
```

```
b := x + y
…
```

Remove unreachable code

Renaming Temporary Variables

- Temporary variables that are dead at the end of a block can be safely renamed
- A statement t : = b + c (t is a temporary) can be changed to u : = b + c (u is a new temporary) and all uses of this instance of t can be changed to u without changing the value of the basic block. Such a block is called a normal-form block.

```
t1 := b + c
t2 := a - t1
t1 := t1 * d
d := t2 + t1
```

```
t1 := b + c
t2 := a - t1
t3 := t1 * d
d := t2 + t3
```

Normal-form block

Interchange of Statements

- Independent statements can be reordered
- Suppose a block has the following two adjacent statements:

  t1 : = b + c

  t2 : = x + y
- We can interchange the two statements without affecting the value of the block if and only if neither x nor y is t1 and neither b nor c is t2.

```
t1 := b + c
t2 := a - t1
t3 := t1 * d
d := t2 + t3
```

⇒

```
t1 := b + c
t3 := t1 * d
t2 := a - t1
d := t2 + t3
```

Note that normal-form blocks permit all statement interchanges that are possible

Algebraic Transformations

- change the set of expressions computed by a basic block into an algebraically equivalent set.
- simplify expressions or replace expensive operations by cheaper ones.
- Example
  - x : = x + 0 or x : = x * 1 can be eliminated from a basic block without changing the set of expressions it computes.
  - The exponential statement x : = y * * 2 can be replaced by x : = y * y.

```
t1 := a - a
t2 := b + t1
t3 := 2 * t2
```

```
t1 := 0
t2 := b
t3 := t2 << 1
```

Next-use Information

- If the name in a register is no longer needed, then we remove the name from the register and the register can be used to store some other names.
- Computing Next Uses
- The *use* of a name in a three-address statement is defined as follows.
  - Suppose three-address statement *i* assigns a value to x. If statement j has x as an operand, and control can flow from statement *i* to j along a path that has no intervening assignments *to* x, then we say statement *j uses* the value of x computed at i .

Next-Use

- Next-use information is needed for dead-code elimination and register assignment
- Next-use is computed by a backward scan of a basic block and performing the following actions on statement

$$i\text{: } x := y \text{ op } z$$

  - Add liveness/next-use info on *x*, *y*, and *z* to statement *i*
  - Set *x* to "not live" and "no next use"
  - Set *y* and *z* to "live" and the next uses of *y* and *z* to *i*

Symbol Table:

Names	Liveliness	Next-use
x	not live	no next-use
y	Live	i
z	Live	i

Next-Use (Step 1)

i: **`a := b + c`**

j: **`t := a + b`** [*live*(**`a`**) = true, *live*(**`b`**) = true, *live*(**`t`**) = true,
nextuse(**`a`**) = none, *nextuse*(**`b`**) = none, *nextuse*(**`t`**) = no

Attach current live/next-use information
Because info is empty, assume variables are live

Next-Use (Step 2)

i: `a := b + c`

live(`a`) = true	*nextuse*(`a`) = *j*
live(`b`) = true	*nextuse*(`b`) = *j*
live(`t`) = false	*nextuse*(`t`) = none

j: `t := a + b` [*live*(`a`) = true, *live*(`b`) = true, *live*(`t`) = true,
nextuse(`a`) = none, *nextuse*(`b`) = none, *nextuse*(`t`) = no

Compute live/next-use information at *j*

Next-Use (Step 3)

i: `a := b + c` [*live*(`a`) = true, *live*(`b`) = true, *live*(`c`) = false,
nextuse(`a`) = *j*, *nextuse*(`b`) = *j*, *nextuse*(`c`) = none]

j: `t := a + b` [*live*(`a`) = true, *live*(`b`) = true, *live*(`t`) = true,
nextuse(`a`) = none, *nextuse*(`b`) = none, *nextuse*(`t`) = no

Attach current live/next-use information to *i*

Next-Use (Step 4)

live(**a**) = false *nextuse*(**a**) = none
live(**b**) = true *nextuse*(**b**) = *i*
live(**c**) = true *nextuse*(**c**) = *i*
live(**t**) = false *nextuse*(**t**) = none

i: **a := b + c** [*live*(**a**) = true, *live*(**b**) = true, *live*(**c**) = false,
nextuse(**a**) = *j*, *nextuse*(**b**) = *j*, *nextuse*(**c**) = none]

j: **t := a + b** [*live*(**a**) = false, *live*(**b**) = false, *live*(**t**) = false,
nextuse(**a**) = none, *nextuse*(**b**) = none, *nextuse*(**t**) = no

Compute live/next-use information *i*

A simple code generator

- A code generator generates target code for a sequence of three- address statements and effectively uses registers to store operands of the statements.(using next-use information)
- Example: consider the three-address statement a := b+c
- It can have the following sequence of codes:

```
ADD Rj, Ri        Cost = 1     // if Ri contains b and Rj contains c
          (or)
ADD c, Ri         Cost = 2     // if c is in a memory location
          (or)
MOV c, Rj         Cost = 3     // move c from memory to Rj and add
ADD Rj, Ri
```

Register and Address Descriptors

- The code-generation algorithm uses descriptors to keep track of register contents and addresses for names.
- A register descriptor is used to keep track of what is currently in each registers. The register descriptors show that initially all the registers are empty.
- An address descriptor stores the location where the current value of the name can be found at run time.

Register and Address Descriptors

- A register descriptor keeps track of what is currently stored in a register at a particular point in the code, e.g. a local variable, argument, global variable, etc.

  MOV a,R0 "R0 contains a"

- An address descriptor keeps track of the location where the current value of the name can be found at run time, e.g. a register, stack location, memory address, etc.

  MOV a,R0

  MOV R0,R1 "a in R0 and R1"

A Code Generator

- Uses new function *getreg* to assign registers to variables
- Computed results are kept in registers as long as possible, which means:
  - Result is needed in another computation
  - Register is kept up to a procedure call or end of block
- Checks if operands to three-address code are available in registers

The Code Generation Algorithm

- For each statement $x := y$ op z
  1. Set location $L = getreg(y, z)$
  2. If $y \notin L$ then generate
     MOV y', L
     where y' denotes one of the locations where the value of y is available (choose register if possible)
  3. Generate
     OP z', L
     where z' is one of the locations of z;
     Update register/address descriptor of x to include L
  4. If y and/or z has no next use and is stored in register, update register descriptors to remove y and/or z

The *getreg* Algorithm

- To compute *getreg*(*y*,*z*)
  1. If *y* is stored in a register *R* and *R* only holds the value *y*, and *y* has no next use, then return *R*;
     Update address descriptor: value *y* no longer in *R*
  2. Else, return a new empty register if available
  3. Else, find an occupied register *R*;
     Store contents (register spill) by generating
     MOV *R***,***M*
     for every *M* in address descriptor of *y*;
     Return register *R*
  4. Return a memory location

Code Generation Example

Ex: d : = (a-b) + (a-c) + (a-c)

3AC:

t : = a – b

u : = a – c

v : = t + u

d : = v + u

with d live at the end.

Statements	*Code Generated*	*Register Descriptor*	*Address Descriptor*
		Registers empty	
`t := a - b`	`MOV a,R0` `SUB b,R0`	**R0** contains **t**	**t** in **R0**
`u := a - c`	`MOV a,R1` `SUB c,R1`	**R0** contains **t** **R1** contains **u**	**t** in **R0** **u** in **R1**
`v := t + u`	`ADD R1,R0`	**R0** contains **v** **R1** contains **u**	**u** in **R1** **v** in **R0**
`d := v + u`	`ADD R1,R0` `MOV R0,d`	**R0** contains **d**	**d** in **R0** **d** in **R0** and memory

- Indexed assignments

Statements	Code Generated	Cost
a : = b[i]	MOV b(R_i), R	2
a[i] : = b	MOV b, a(R_i)	3

- Pointer assignments

Statements	Code Generated	Cost
a : = *p	MOV *R_p, a	2
*p : = a	MOV a, *R_p	2

- Conditional assignments

Statement	Code
if x < y goto z	CMP x, y CJ< z /* jump to z if condition code is negative */

x : = y +z
if x < 0 goto z
	MOV y, R_0
	ADD z, R_0
	MOV R_0,x
	CJ< z

Register Allocation and Assignment

- The *getreg* algorithm is simple but sub-optimal
  - All live variables in registers are stored (flushed) at the end of a block
- *Global register allocation* assigns variables to limited number of available registers and attempts to keep these registers consistent across basic block boundaries
  - Keeping variables in registers in looping code can result in big savings

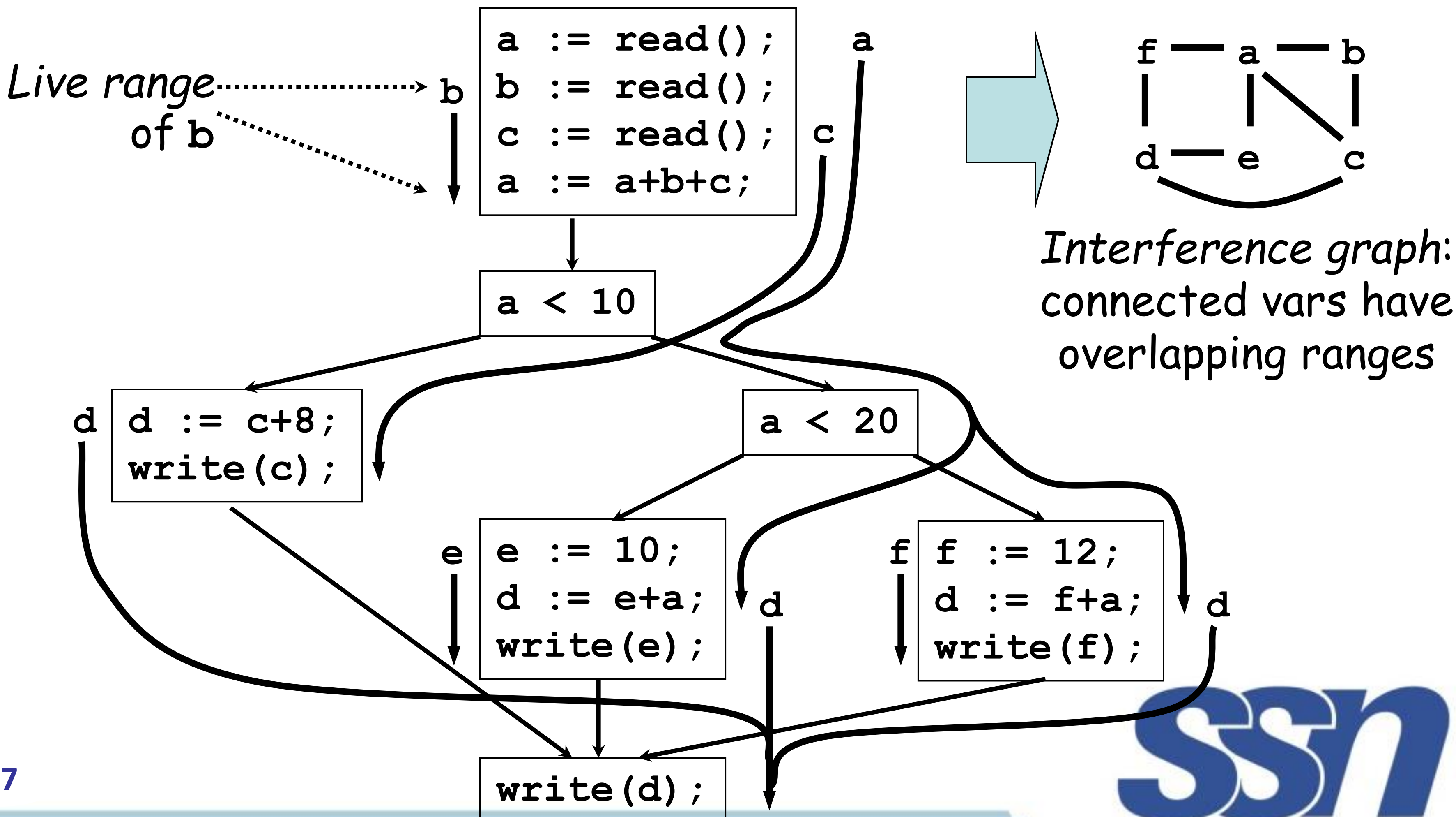
Global Register Allocation with Graph Coloring

- When a register is needed but all available registers are in use, the content of one of the used registers must be stored (spilled) to free a register
- Graph coloring allocates registers and attempts to minimize the cost of spills
- Build a *conflict graph* (*interference graph*)
- Find a *k*-coloring for the graph, with *k* the number of registers

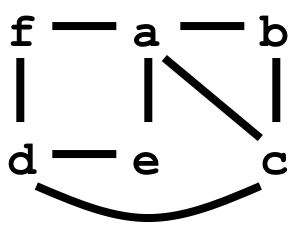
Register Allocation with Graph Coloring: Example

```
a := read();
b := read();
c := read();
a := a + b + c;
if (a < 10) {
    d := c + 8;
    write(c);
} else if (a < 20) {
    e := 10;
    d := e + a;
    write(e);
} else {
    f := 12;
    d := f + a;
    write(f);
}
write(d);
```

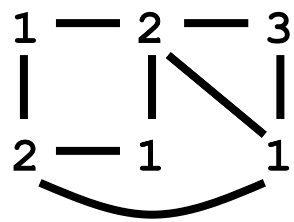
Register Allocation with Graph Coloring: Live Ranges



Register Allocation with Graph Coloring: Solution



Interference graph Solve



Three registers:

```
a = r2
b = r3
c = r1
d = r2
e = r1
f = r1
```

```
r2 := read();
r3 := read();
r1 := read();
r2 := r2 + r3 + r1;
if (r2 < 10) {
    r2 := r1 + 8;
    write(r1);
} else if (r2 < 20) {
    r1 := 10;
    r2 := r1 + r2;
    write(r1);
} else {
    r1 := 12;
    r2 := r1 + r2;
    write(r1);
}
write(r2);
```

Peephole Optimization

- A statement-by-statement code-generations strategy often produce target code that contains redundant instructions and suboptimal constructs .The quality of such target code can be improved by applying “optimizing” transformations to the target program.
- Examines a short sequence of target instructions in a window (peephole) and replaces the instructions by a faster and/or shorter sequence when possible
- The peephole is a small, moving window on the target program.
- Applied to intermediate code or target code
- Typical optimizations:
  - Redundant instruction elimination
  - Flow-of-control optimizations
  - Algebraic simplifications
  - Use of machine idioms

Peephole Opt: Eliminating Redundant Loads and Stores

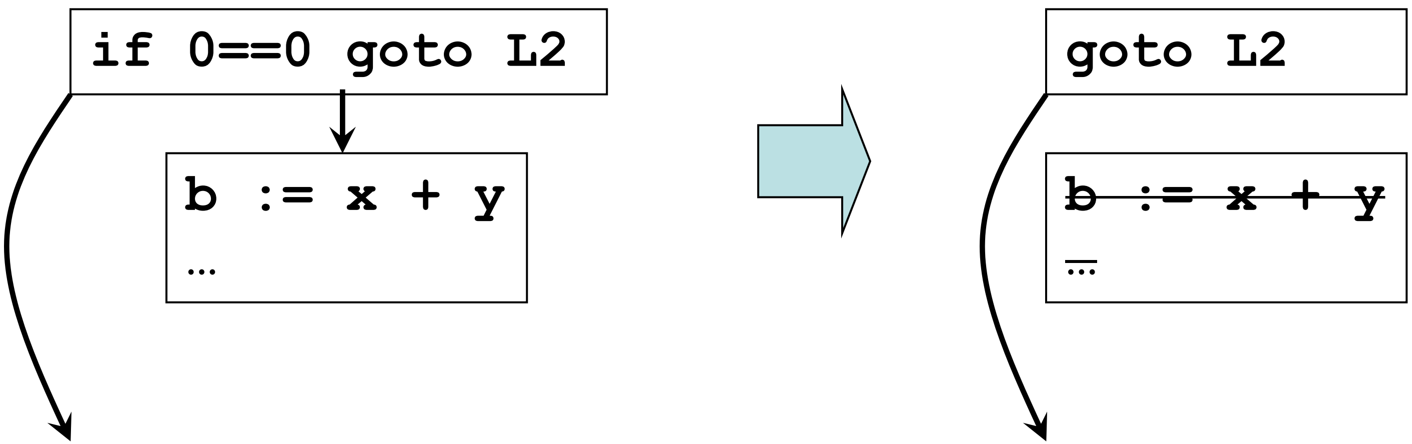
- Consider

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

- The second instruction can be deleted, but only if it is not labeled with a target label
  - Peephole represents sequence of instructions with at most one entry point
- The first instruction can also be deleted if *live*(**a**)=false

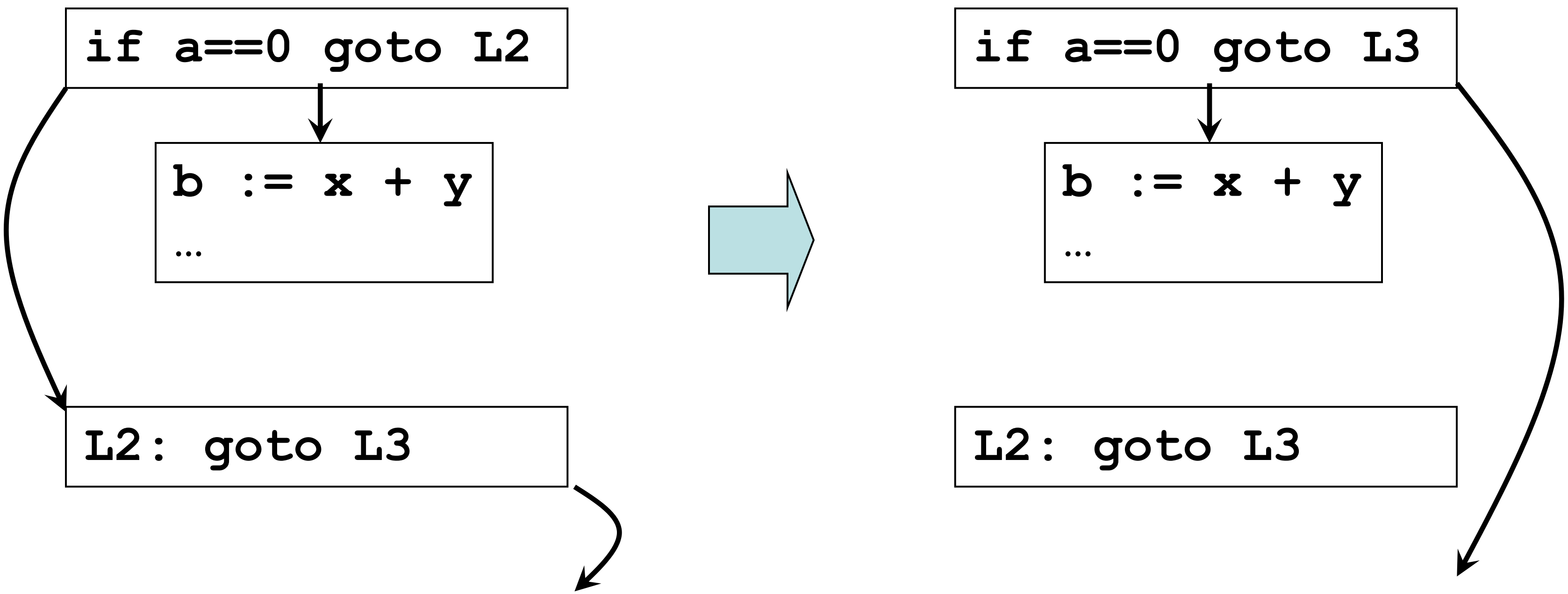
Peephole Optimization: Deleting Unreachable Code

- Unlabeled blocks can be removed



Peephole Optimization: Branch Chaining

- Shorten chain of branches by modifying target labels



Peephole Optimization: Other Flow-of-Control Optimizations

- Remove redundant jumps

```
…
goto L1
```

```
L1:
…
```

```
…
```

Other Peephole Optimizations

- *Reduction in strength*: replace expensive arithmetic operations with cheaper ones

```
…
a := x ^ 2
b := y / 8
```

⇨

```
…
a := x * x
b := y >> 3
```

- Utilize machine idioms

```
…
a := a + 1
```

⇨

```
…
inc a
```

- Algebraic simplifications

```
…
a := a + 0
b := b * 1
```

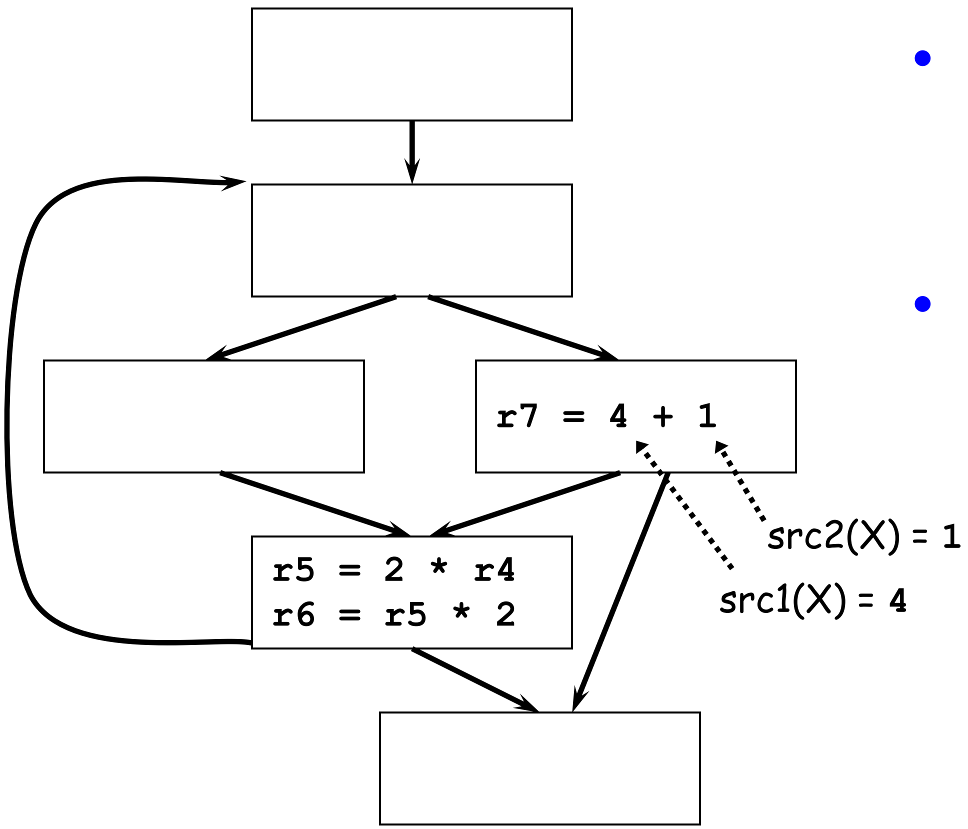
⇨

```
…
```

Classic Examples of Local and Global Code Optimizations

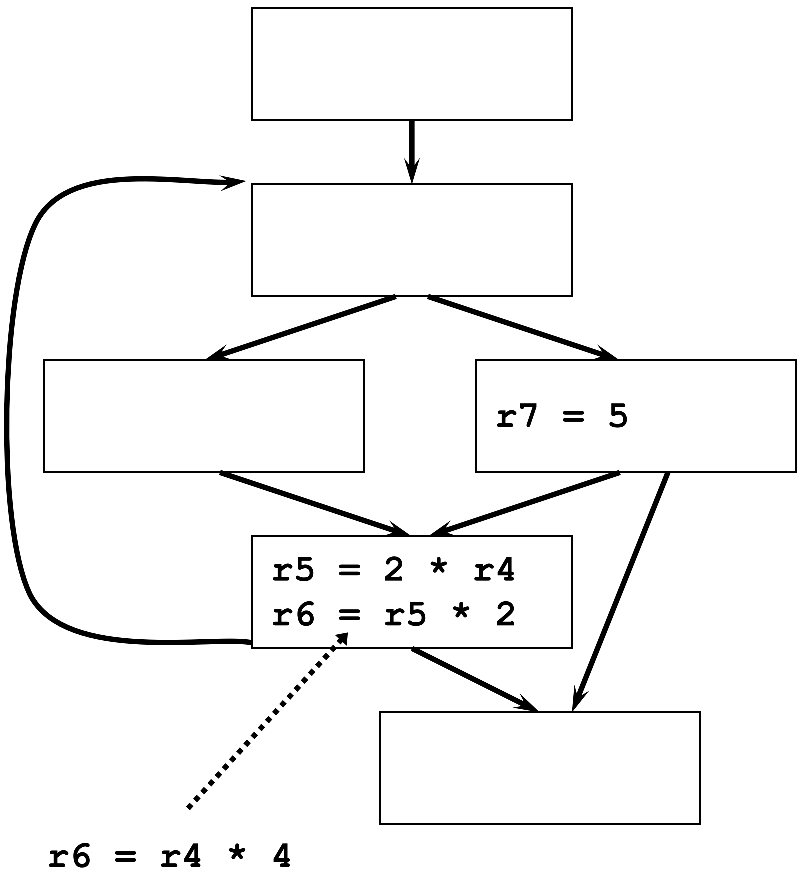
- Local
  - Constant folding
  - Constant combining
  - Strength reduction
  - Constant propagation
  - Common subexpression elimination
  - Backward copy propagation
- Global
  - Dead code elimination
  - Constant propagation
  - Forward copy propagation
  - Common subexpression elimination
  - Code motion
  - Loop strength reduction
  - Induction variable elimination

Local: Constant Folding



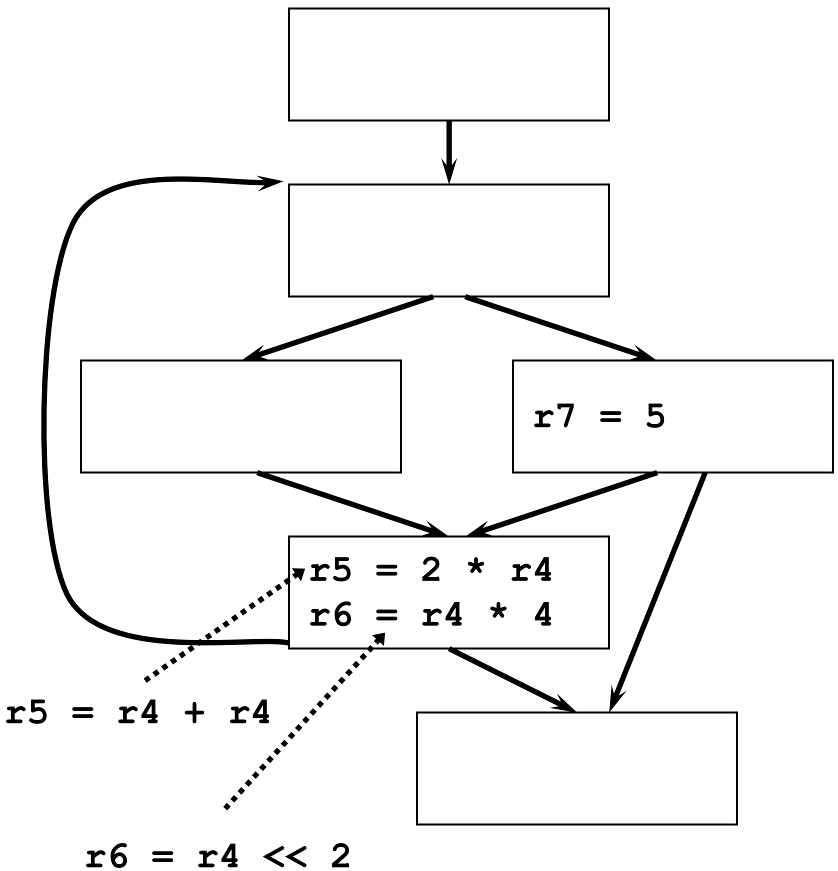
- Goal: eliminate unnecessary operations
- Rules:
  1. X is an arithmetic operation
  2. If src1(X) and src2(X) are constant, then change X by applying the operation

Local: Constant Combining



- Goal: eliminate unnecessary operations
  - First operation often becomes dead after constant combining
- Rules:
  1. Operations X and Y in same basic block
  2. X and Y have at least one literal src
  3. Y uses dest(X)
  4. None of the srcs of X have defs between X and Y (excluding Y)

Local: Strength Reduction



- Goal: replace expensive operations with cheaper ones
- Rules (common):
  1. X is an multiplication operation where src1(X) or src2(X) is a const 2^k integer literal
  2. Change X by using shift operation
  3. For k=1 can use add

Local: Constant Propagation

```
r1 = 5
r2 = _x
r3 = 7
r4 = r4 + r1  <····  r4 = r4 + 5
r1 = r1 + r2  <····  r1 = 5 + _x
r1 = r1 + 1   <····  r1 = 5 + _x + 1
r3 = 12
r8 = r1 - r2  <····  r8 = 5 + _x + 1 - _x
r9 = r3 + r5  <····  r9 = 12 + r5
r3 = r2 + 1   <····  r3 = _x + 1
r7 = r3 - r1  <····  r7 = _x + 1 - 5 - _x - 1
M[r7] = 0
```

- Goal: replace register uses with literals (constants) in a single basic block
- Rules:
  1. Operation X is a move to register with src1(X) literal
  2. Operation Y uses dest(X)
  3. There is no def of dest(X) between X and Y (excluding defs at X and Y)
  4. Replace dest(X) in Y with src1(X)

Local: Common Subexpression Elimination (CSE)

```
r1 = r2 + r3
r4 = r4 + 1
r1 = 6
r6 = r2 + r3
r2 = r1 - 1
r5 = r4 + 1
r7 = r2 + r3
r5 = r1 - 1   <---- r5 = r2
```

- Goal: eliminate re-computations of an expression
  - More efficient code
  - Resulting moves can get copy propagated (see later)
- Rules:
  1. Operations X and Y have the same opcode and Y follows X
  2. src(X) = src(Y) for all srcs
  3. For all srcs, no def of a src between X and Y (excluding Y)
  4. No def of dest(X) between X and Y (excluding X and Y)
  5. Replace Y with move dest(Y) = dest(X)

Local: Backward Copy Propagation

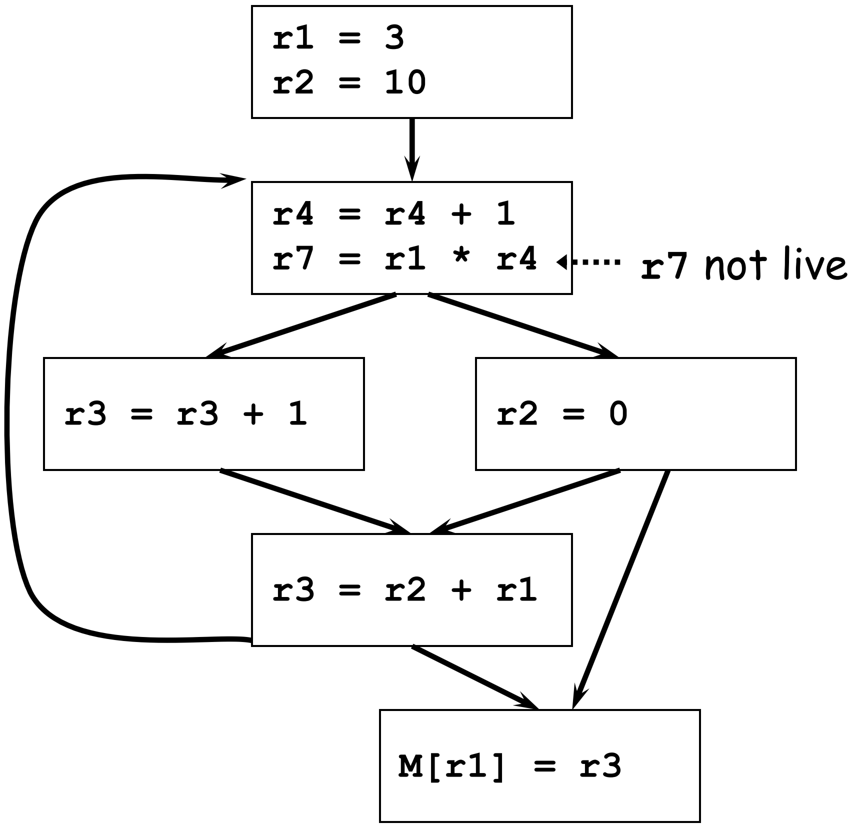
```
r1 = r8 + r9
r2 = r9 + r1
r4 = r2
r6 = r2 + 1     <···· r7 = r2 + 1
r9 = r1
r7 = r6         <···· remove r7 = r6
r5 = r7 + 1
r4 = 0
r8 = r2 + r7
```

r6 not live

- Goal: propagate LHS of moves backward
  - Eliminates useless moves
- Rules (dataflow required)

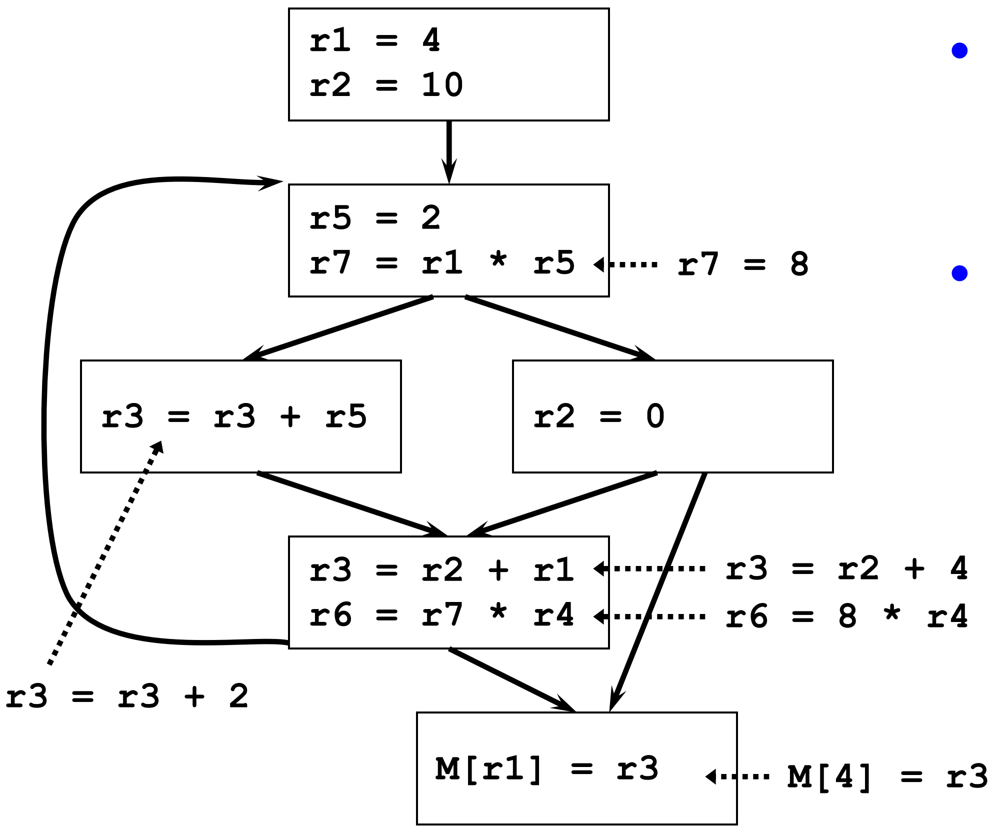
1. X and Y in same block
2. Y is a move to register
3. dest(X) is a register that is not live out of the block
4. Y uses dest(X)
5. dest(Y) not used or defined between X and Y (excluding X and Y)
6. No uses of dest(X) after the first redef of dest(Y)
7. Replace src(Y) on path from X to Y with dest(X) and remove Y

Global: Dead Code Elimination



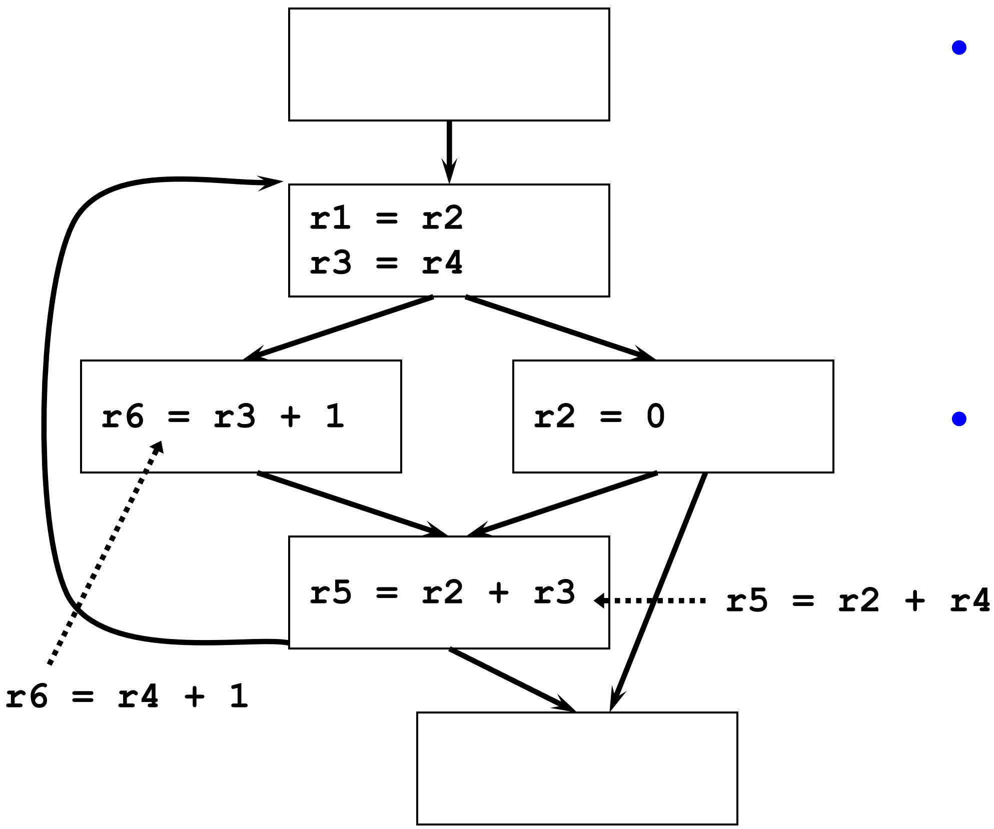
- Goal: eliminate any operation who's result is never used
- Rules (dataflow required)
  1. X is an operation with no use in def-use (DU) chain, i.e. dest(X) is not live
  2. Delete X if removable (not a mem store or branch)
- Rules too simple!
  – Misses deletion of **r4**, even after deleting **r7**, since **r4** is live in loop
  – Better is to trace UD chains backwards from "critical" operations

Global: Constant Propagation



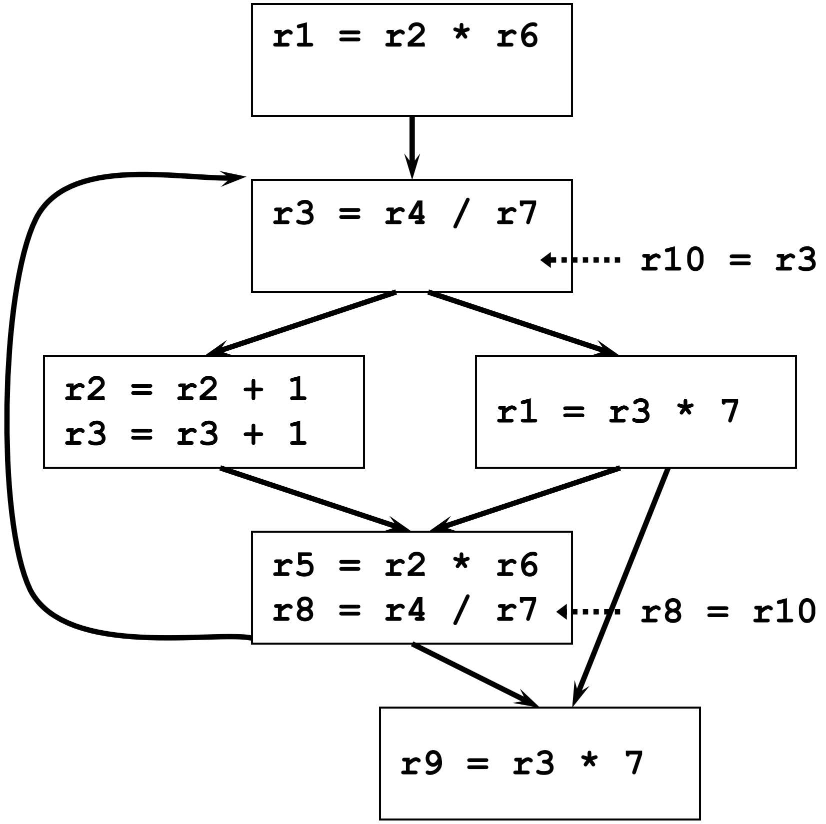
- Goal: globally replace register uses with literals
- Rules (dataflow required)
  1. X is a move to a register with src1(X) literal
  2. Y uses dest(X)
  3. dest(X) has only one def at X for use-def (UD) chains to Y
  4. Replace dest(X) in Y with src1(X)

Global: Forward Copy Propagation



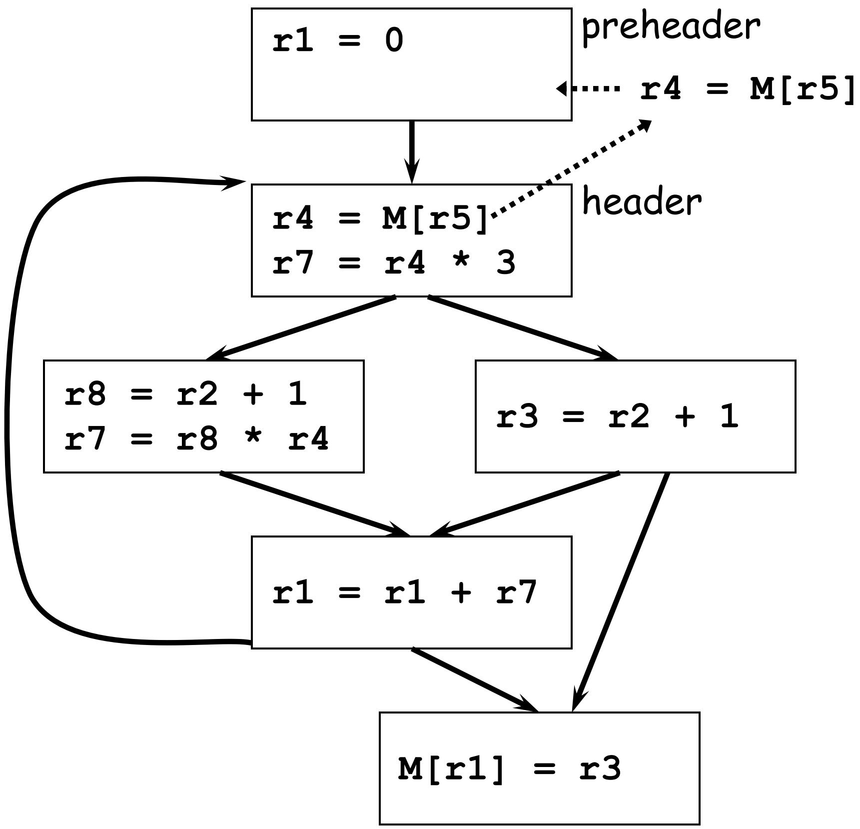
- Goal: globally propagate RHS of moves forward
  - Reduces dependence chain
  - May be possible to eliminate moves
- Rules (dataflow required)
  1. X is a move with src1(X) register
  2. Y uses dest(X)
  3. dest(X) has only one def at X for UD chains to Y
  4. src1(X) has no def on any path from X to Y
  5. Replace dest(X) in Y with src1(X)

Global: Common Subexpression Elimination (CSE)



- Goal: eliminate recomputations of an expression
- Rules:
  1. X and Y have the same opcode and X dominates Y
  2. src(X) = src(Y) for all srcs
  3. For all srcs, no def of a src on any path between X and Y (excluding Y)
  4. Insert rx = dest(X) immediately after X for new register rx
  5. Replace Y with move dest(Y) = rx

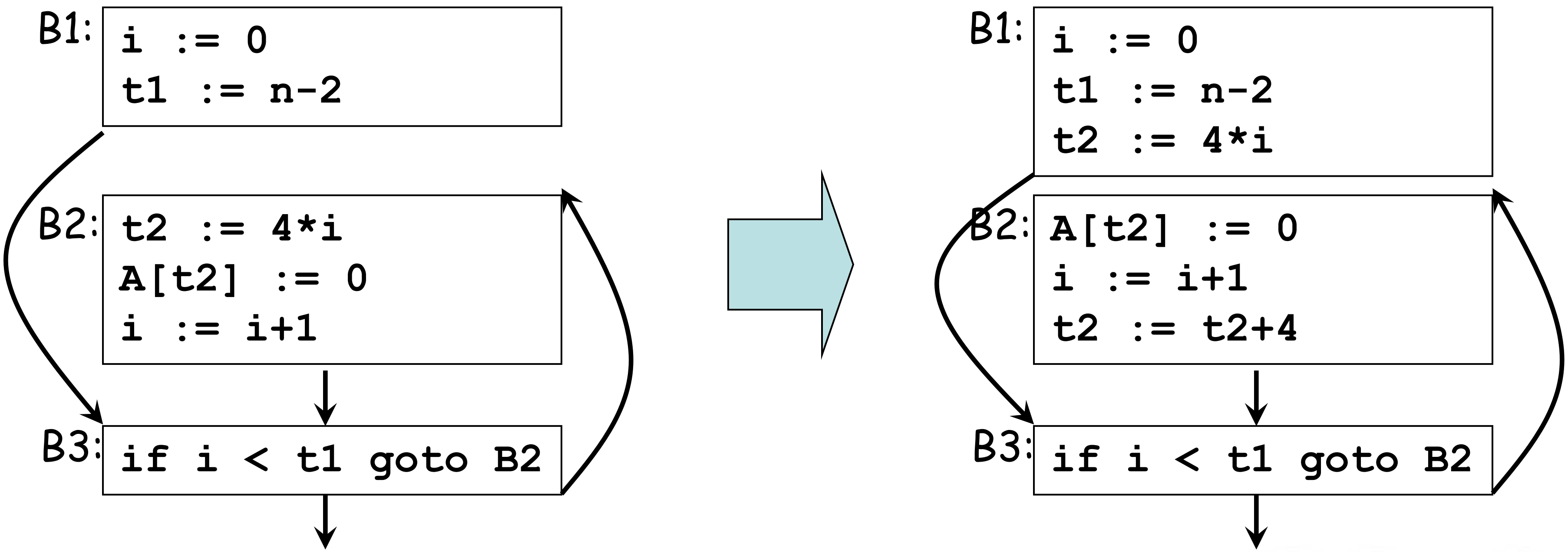
Global: Code Motion



- Goal: move loop-invariant computations to preheader
- Rules:

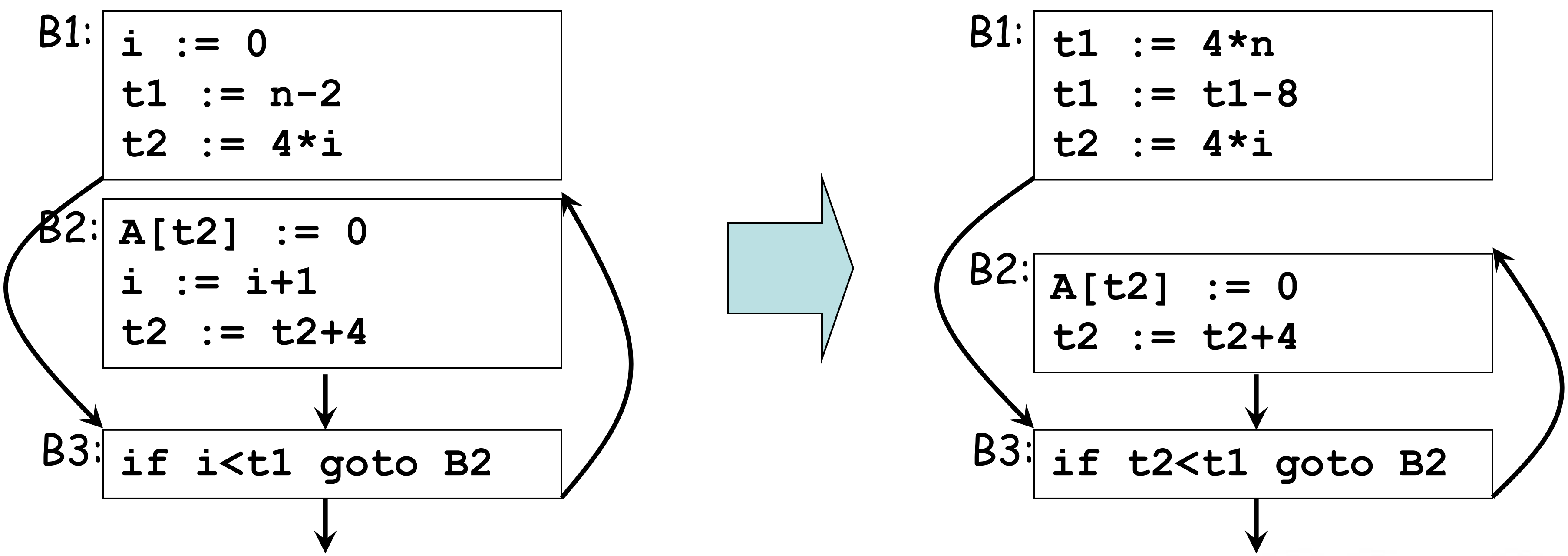
1. Operation X in block that dominates all exit blocks
2. X is the only operation to modify dest(X) in loop body
3. All srcs of X have no defs in any of the basic blocks in the loop body
4. Move X to end of preheader
5. Note 1: if one src of X is a memory load, need to check for stores in loop body
6. Note 2: X must be movable and not cause exceptions

Global: Loop Strength Reduction



Replace expensive computations with *induction variables*

Global: Induction Variable Elimination



Replace induction variable in expressions with another

References

- John E Hopcroft and Jeffery D Ullman, Introduction to Automata Theory, Languages and Computations, Narosa Publishing House, 2002.
- Michael Sipser, "Introduction of the Theory of Computation", Second Edition, Thomson Brokecole, 2006.
- J. Martin, "Introduction to Languages and the Theory of Computation", Third Edition, Tata McGraw Hill, 2003.