Code Optimization

Course : 2CS701 – Compiler Construction

Prof Monika Shah

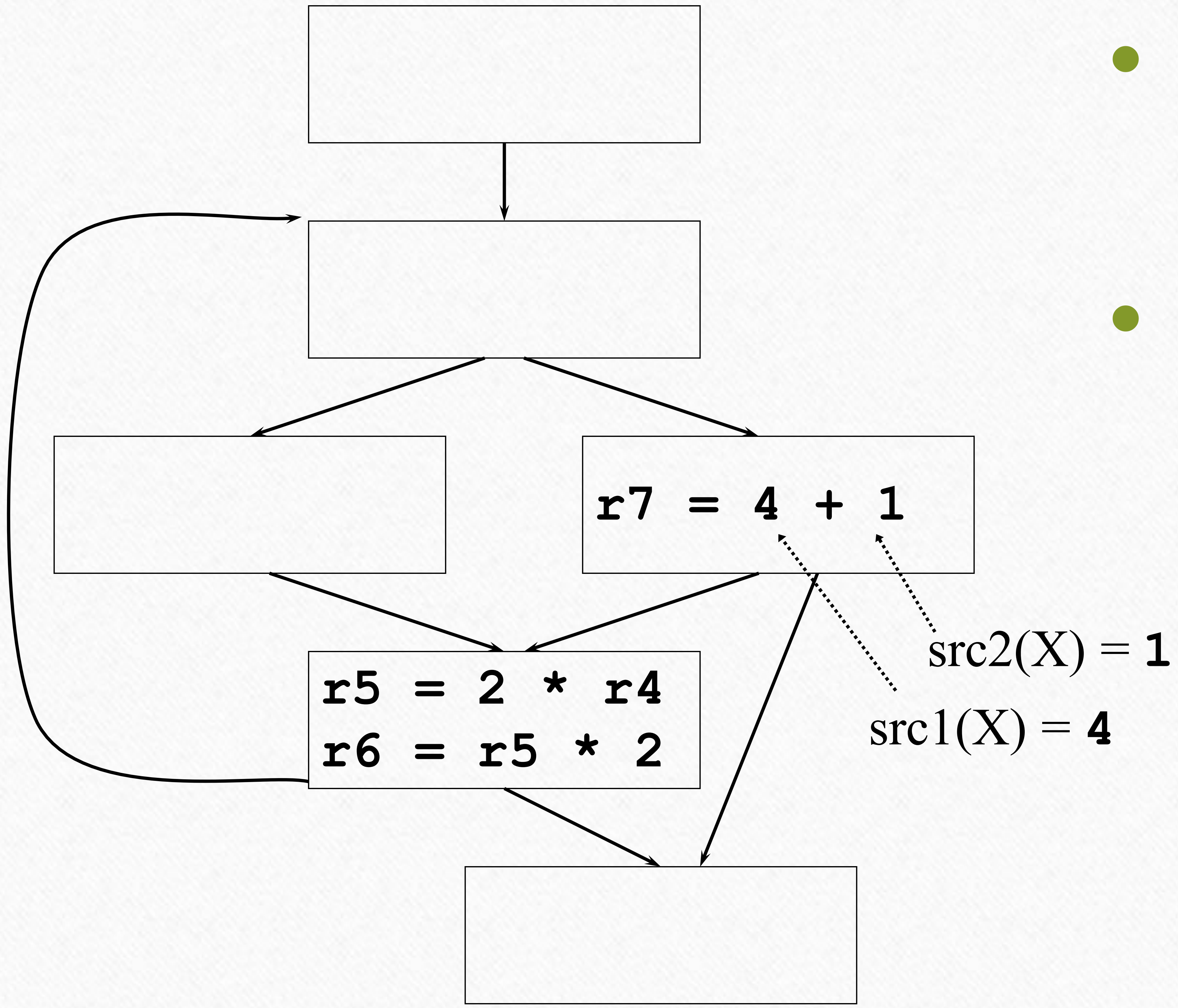
Nirma University

Ref : Ch.9 Compilers Principles, Techniques, and Tools by Alfred Aho, Ravi Sethi, and Jeffrey Ullman (1st Edition

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

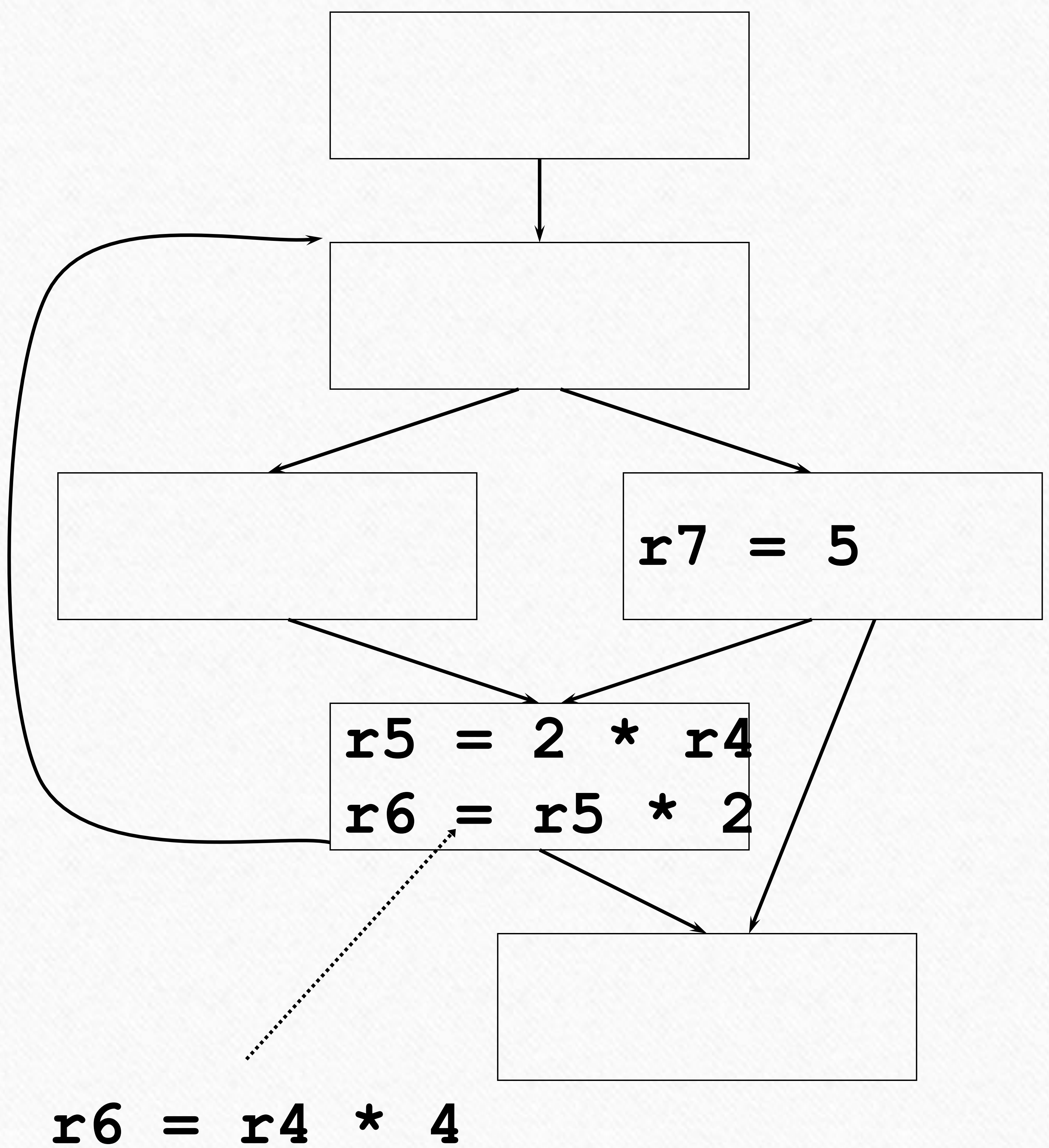
Local: Constant Folding



- Goal: eliminate unnecessary operations
- Rules:

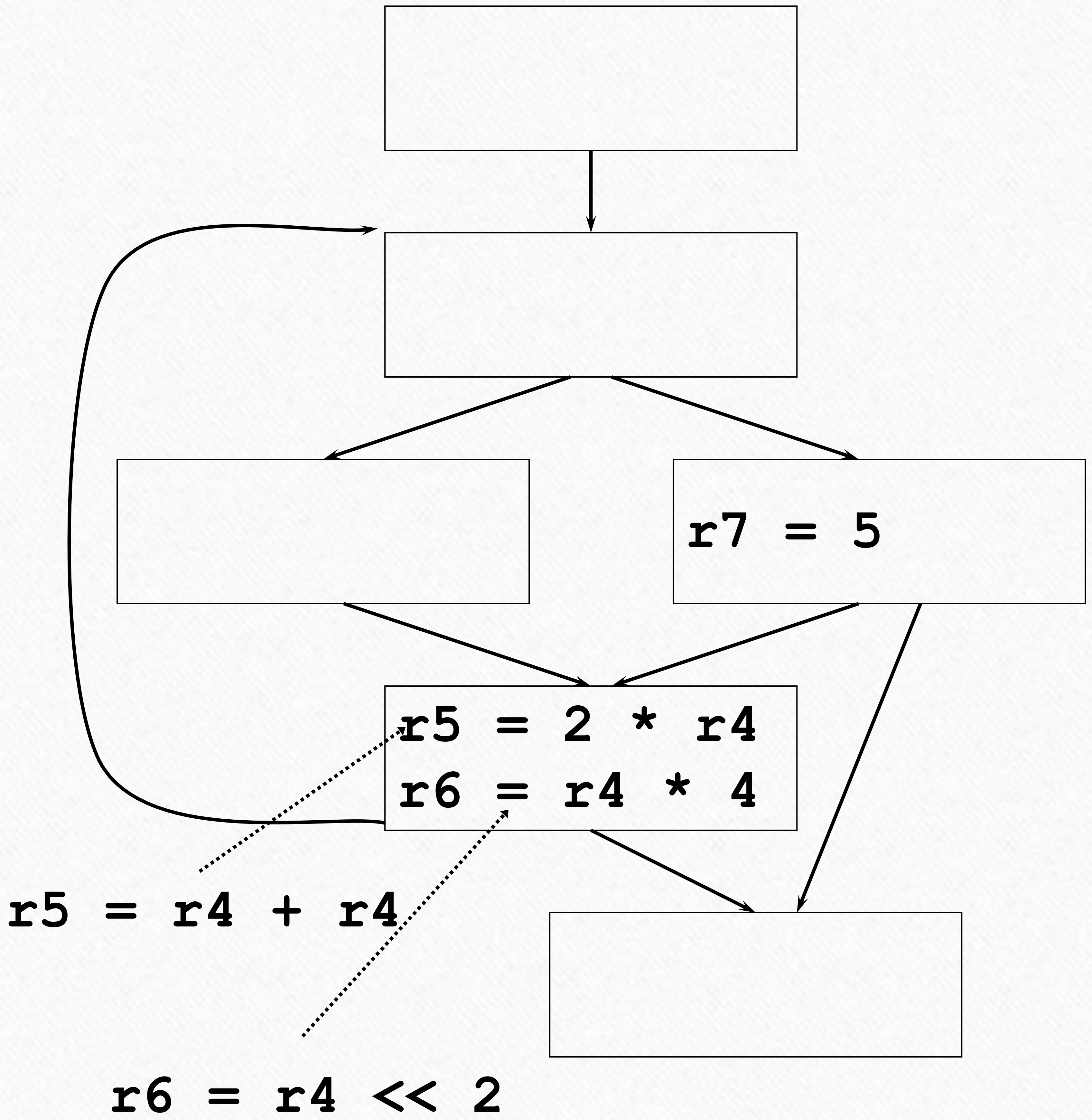
1. X is an arithmetic operation
2. If src1(X) and src2(X) are constant, then change X by applying the operation

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      <-- r5 = r2
r7 = r2 + r3
r5 = r1 - 1
```

- 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

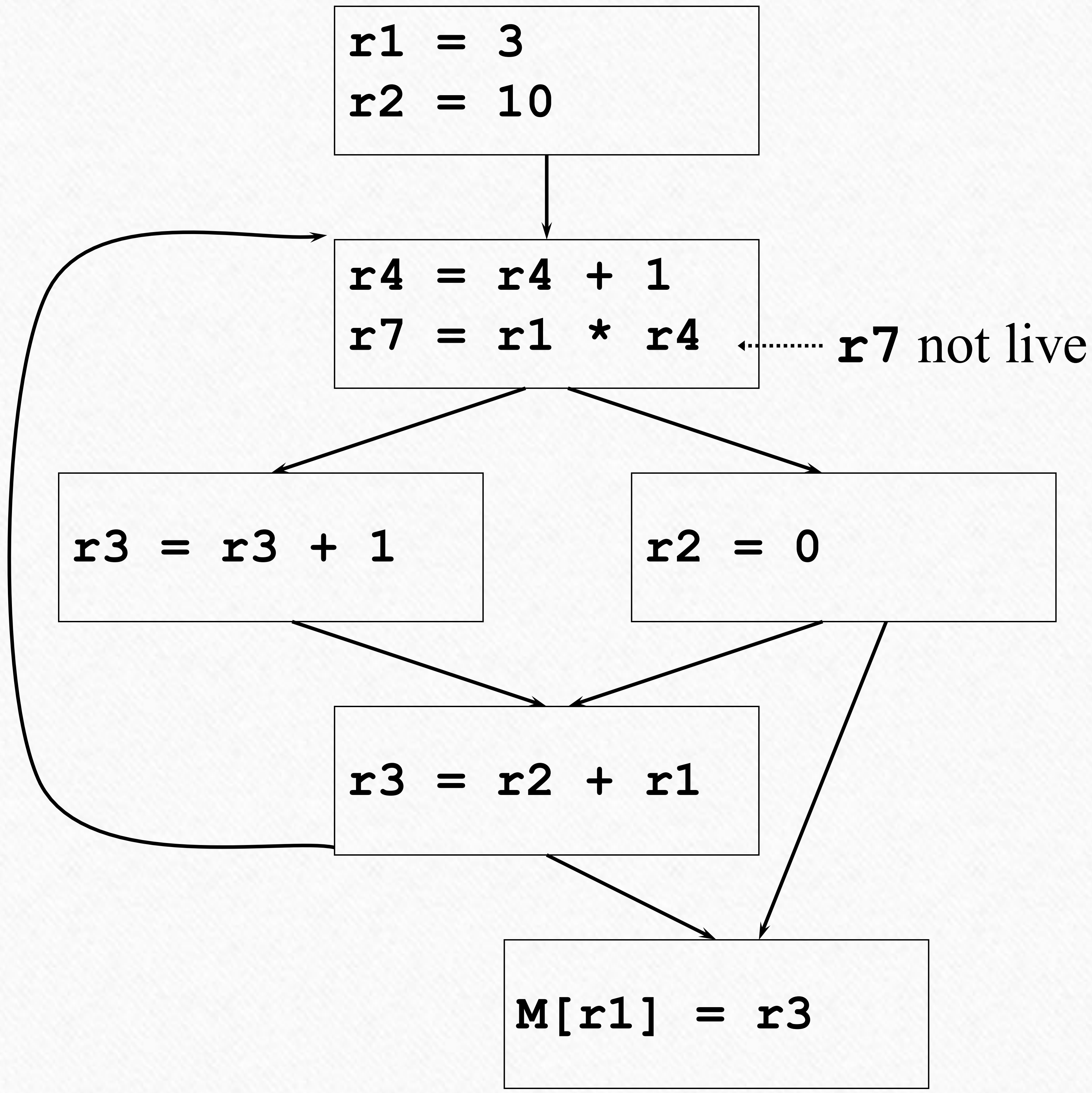
```
r1 = r8 + r9
r2 = r9 + r1
r4 = r2
r6 = r2 + 1      ← r7 = r2 + 1
r9 = r1
r7 = r6          ← remove r7 = r6
r5 = r7 + 1
r4 = 0
r8 = r2 + r7
```

`r6` not live

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

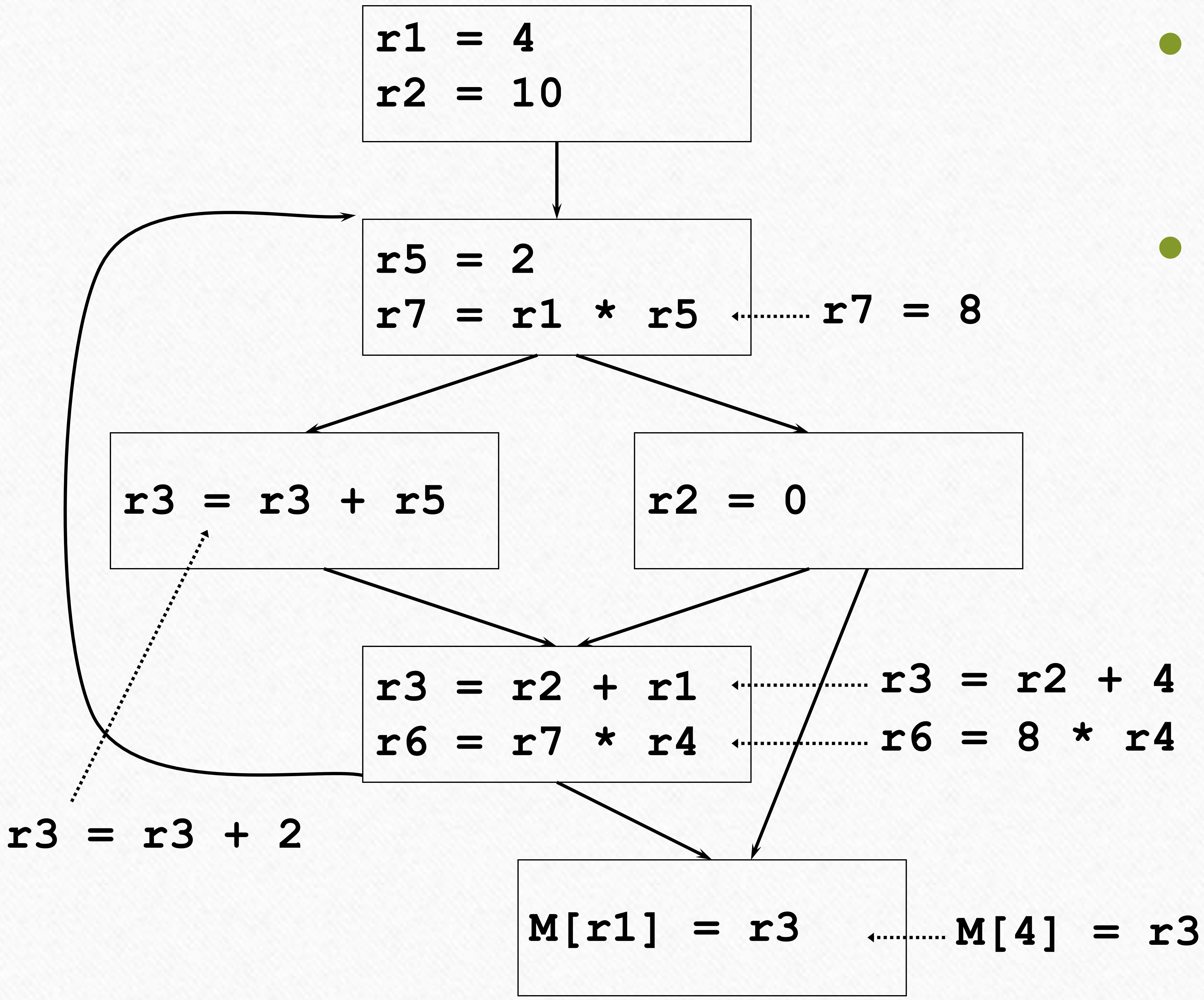
1. X and Y in same block
2. Y is a move to register
3. dest(X) is a register that is not live out of the block
4. Y uses dest(X)
5. dest(Y) not used or defined between X and Y (excluding X and Y)
6. No uses of dest(X) after the first redef of dest(Y)
7. Replace src(Y) on path from X to Y with dest(X)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

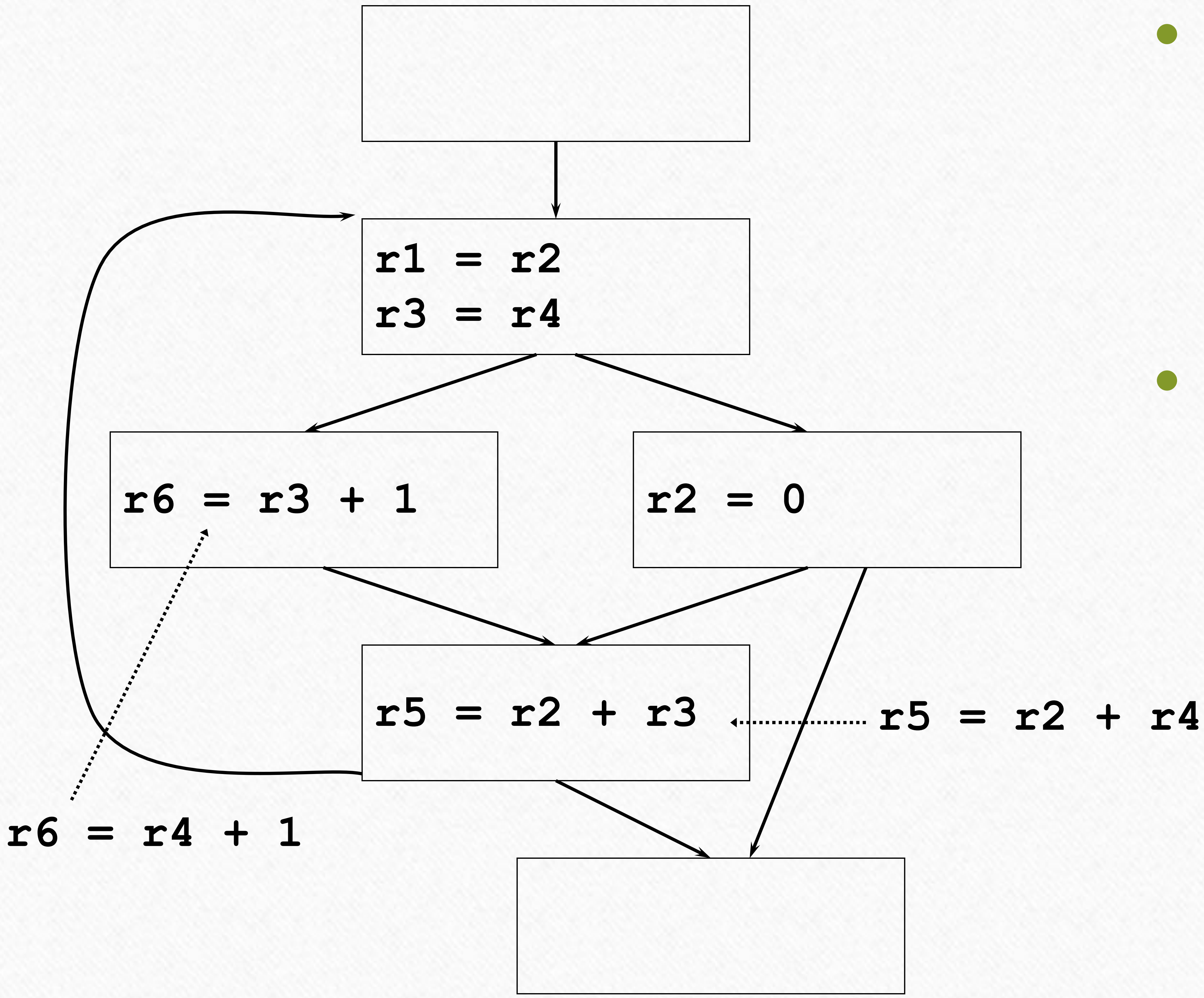
Global: Constant Propagation



- 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)

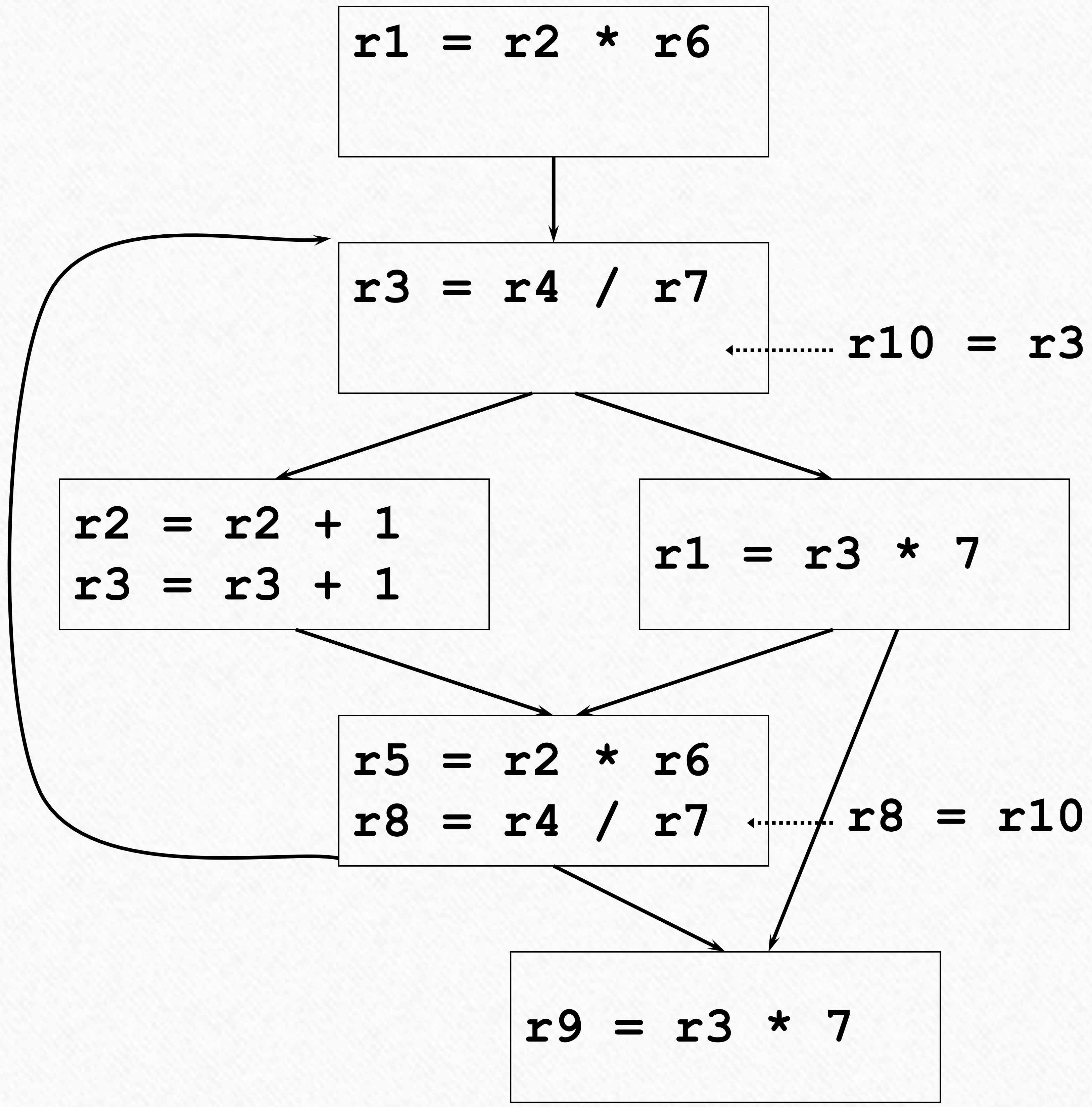
Global: Forward Copy Propagation



- 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)

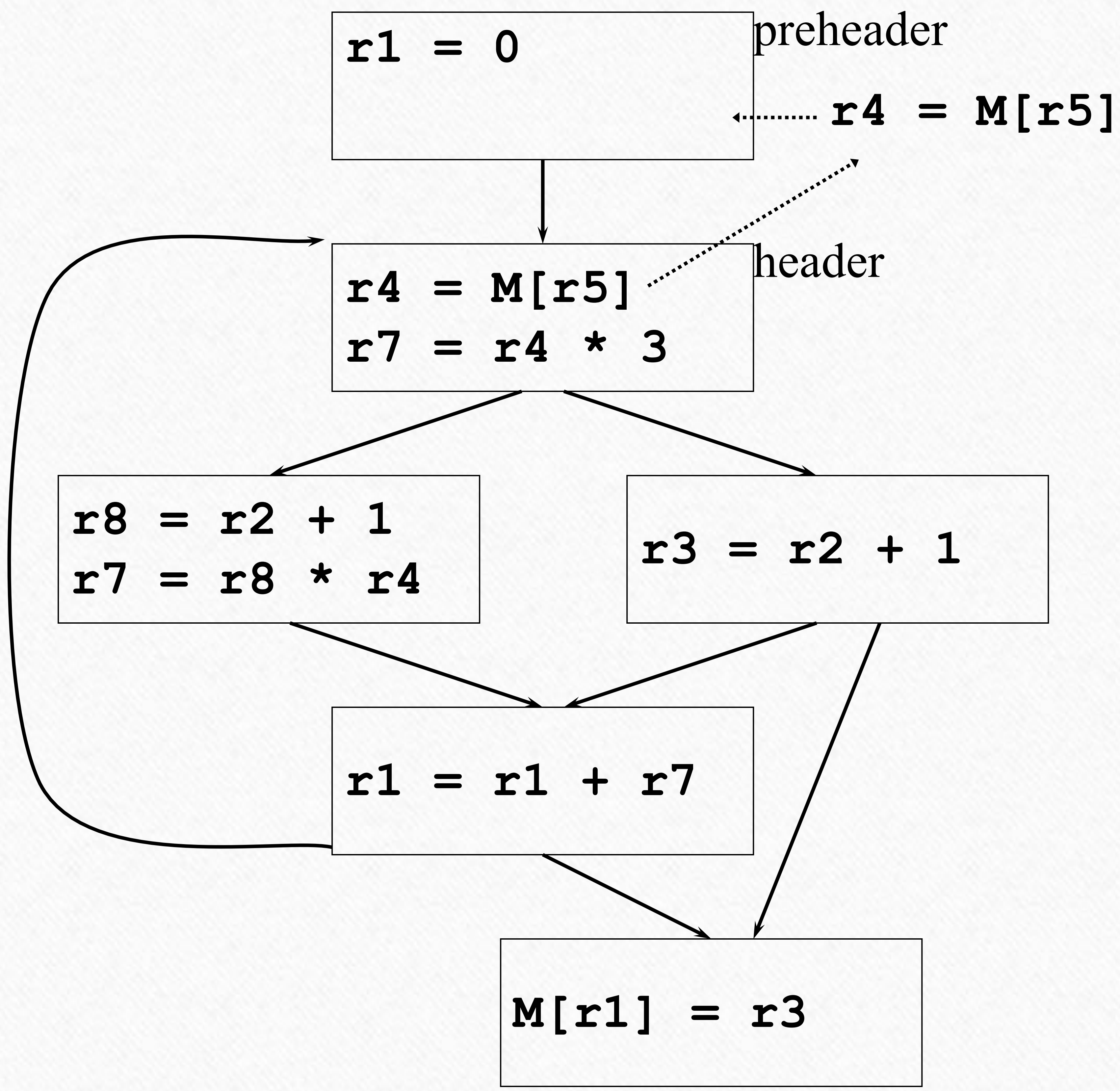
Global: Common Subexpression Elimination (CSE)



- 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

```
B1: i  := 0
    t1  := n-2

B2: t2  := 4*i
    A[t2]  := 0
    i  := i+1

B3: if i < t1 goto B2
```

```
B1: i  := 0
    t1  := n-2
    t2  := 4*i

B2: A[t2]  := 0
    i  := i+1
    t2  := t2+4

B3: if i < t1 goto B2
```


Replace expensive computations with *induction variables*

Global: Induction Variable Elimination

```
B1: i  := 0
    t1 := n-2
    t2 := 4*i

B2: A[t2] := 0
    i  := i+1
    t2 := t2+4

B3: if i<t1 goto B2
```

```
B1: t1 := 4*n
    t1 := t1-8
    t2 := 4*i

B2: A[t2] := 0
    t2 := t2+4

B3: if t2<t1 goto B2
```


Replace induction variable in expressions with another

Peephole Optimization

- Examines a short sequence(usually contiguous) of target instructions in a window (*peephole*) and replaces the instructions by a faster and/or shorter sequence when possible
- Effective for improving assembly code
- Typical optimizations:
    - Redundant instruction elimination
    - Flow-of-control optimizations
    - Algebraic simplifications
    - Use of machine idioms

Peephole Optimization

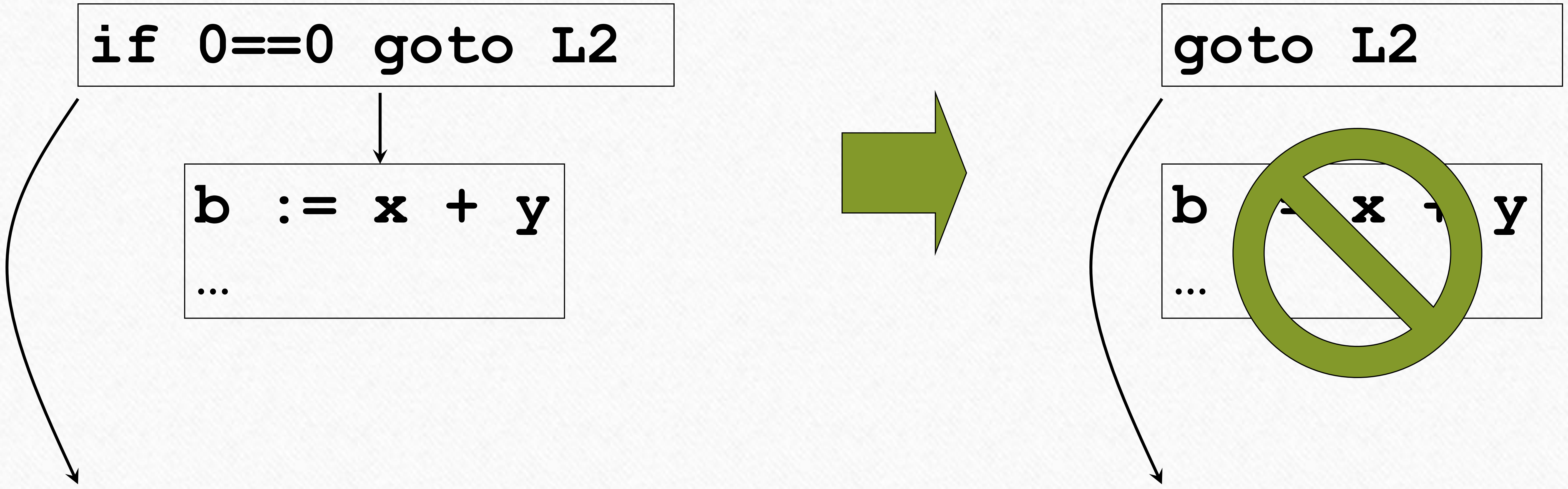
- Examines a short sequence(usually contiguous) of target instructions in a window (*peephole*) and replaces the instructions by a faster and/or shorter sequence when possible
- Effective for improving assembly code
- Objectives:
  - Improve performance
  - Reduce memory footprint
  - Reduce code size

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

```
if a==0 goto L2

b := x + y
…

L2: goto L3
```

```
if a==0 goto L3

b := x + y
…

L2: goto L3
```

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
```

➡

```
…
```