Topic 11: Loops

COS 320

Compiling Techniques

Princeton University
Spring 2016

Lennart Beringer

Loop Preheaders

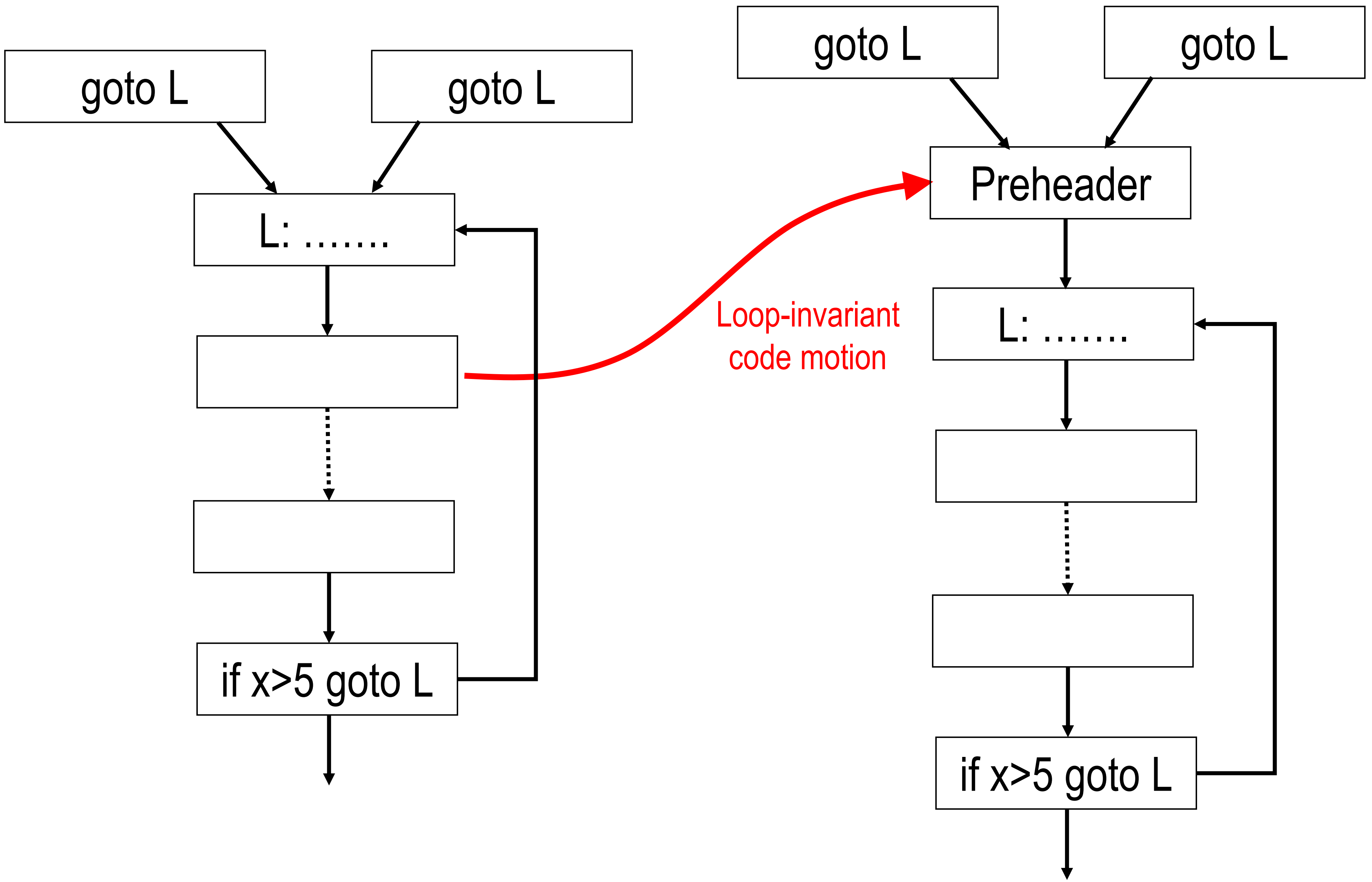
Recall:

- A *loop* is a set of CFG nodes S such that:
  1. there exists a *header* node h in S that dominates all nodes in S.
     - there exists a path of directed edges from h to any node in S.
     - h is the only node in S with predecessors not in S.
  2. from any node in S, there exists a path of directed edges to h.
- A loop is a single entry, multiple exit region.

Loop Preheaders:

- Some loop optimizations (loop invariant code removal) need to insert statements immediately before loop header.
- Create a loop *preheader* - a basic block before the loop header block.

Loop Preheader Example



Loop Invariant Computation

- Given statements in loop s: $\texttt{t = } a_1 \texttt{ op } a_2$:
  - s is loop-invariant if a_1, a_2 have same value each loop iteration.
  - may sometimes be possible to hoist s outside loop.
- Cannot always tell whether a will have same value each iteration $\rightarrow$ conservative approximation.
- d: $\texttt{t = } a_1 \texttt{ op } a_2$ is loop-invariant within loop L if for each a_i:
  1. a_i is constant, or
  2. all definitions of a_i that reach d are outside L, or
  3. only one definition of a_i reaches d, and is loop-invariant.

Loop Invarient Computation

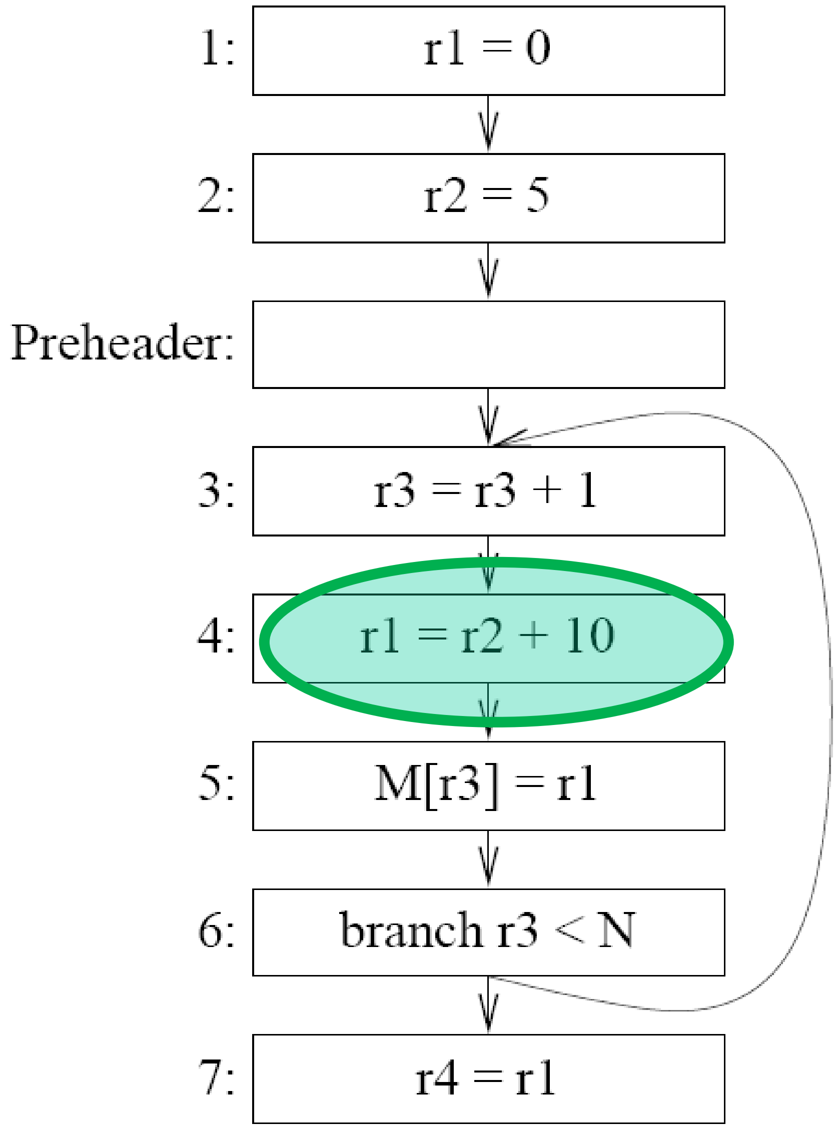
Iterative algorithm for determining loop-invariant computations:

```
mark "invariant" all definitions whose operands
- are constant, or
- whose reaching definitions are outside loop.

WHILE (changes have occurred)
  mark "invariant" all definitions whose operands
  - are constant,
  - whose reaching definitions are outside loop, or
  - which have a single reaching definition in loop
    marked invariant.
```

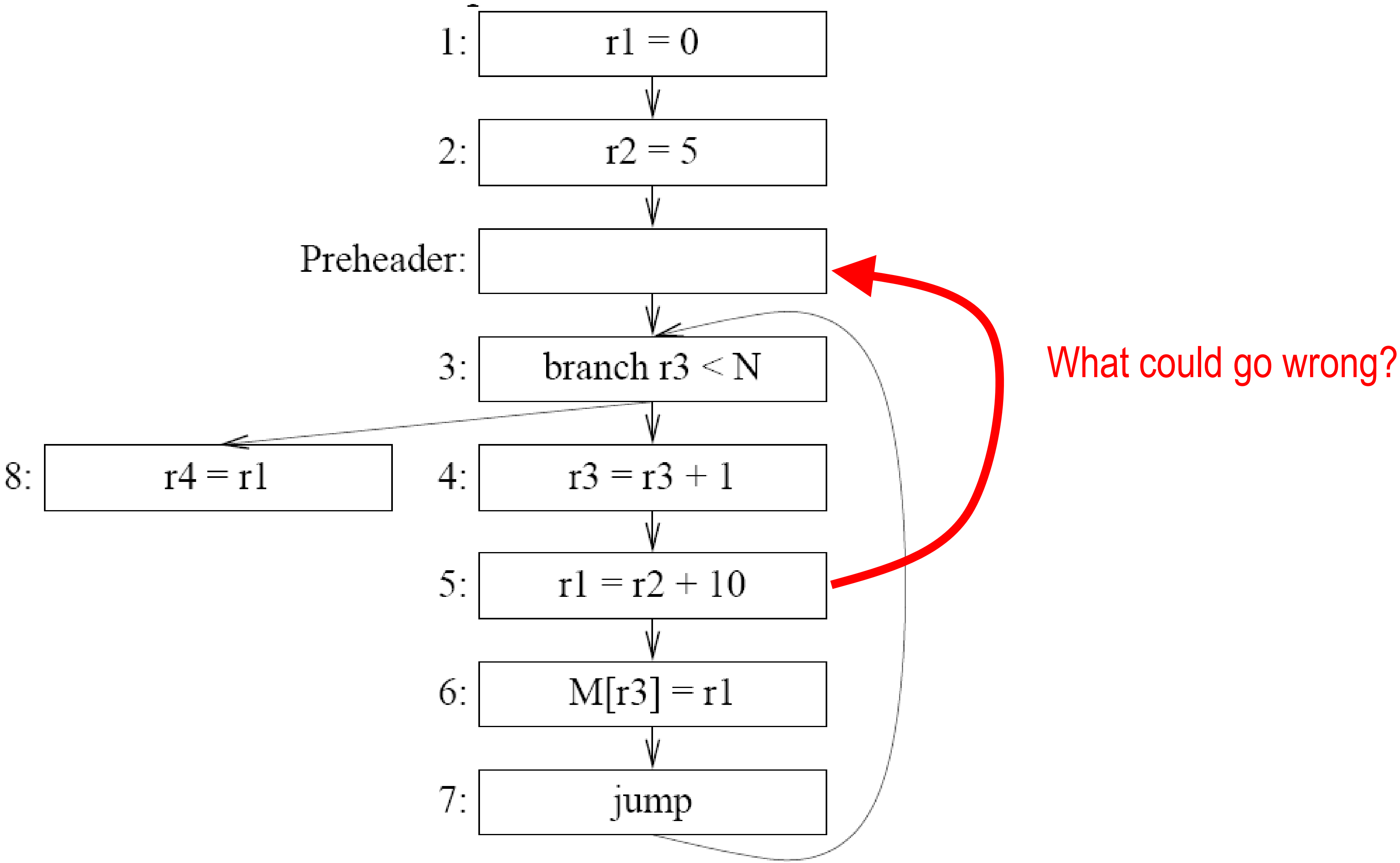
Loop Invariant Code Motion (LICM)

After detecting loop-invariant computations, perform code motion.



Subject to some constraints.

LICM: motivating constraint 1

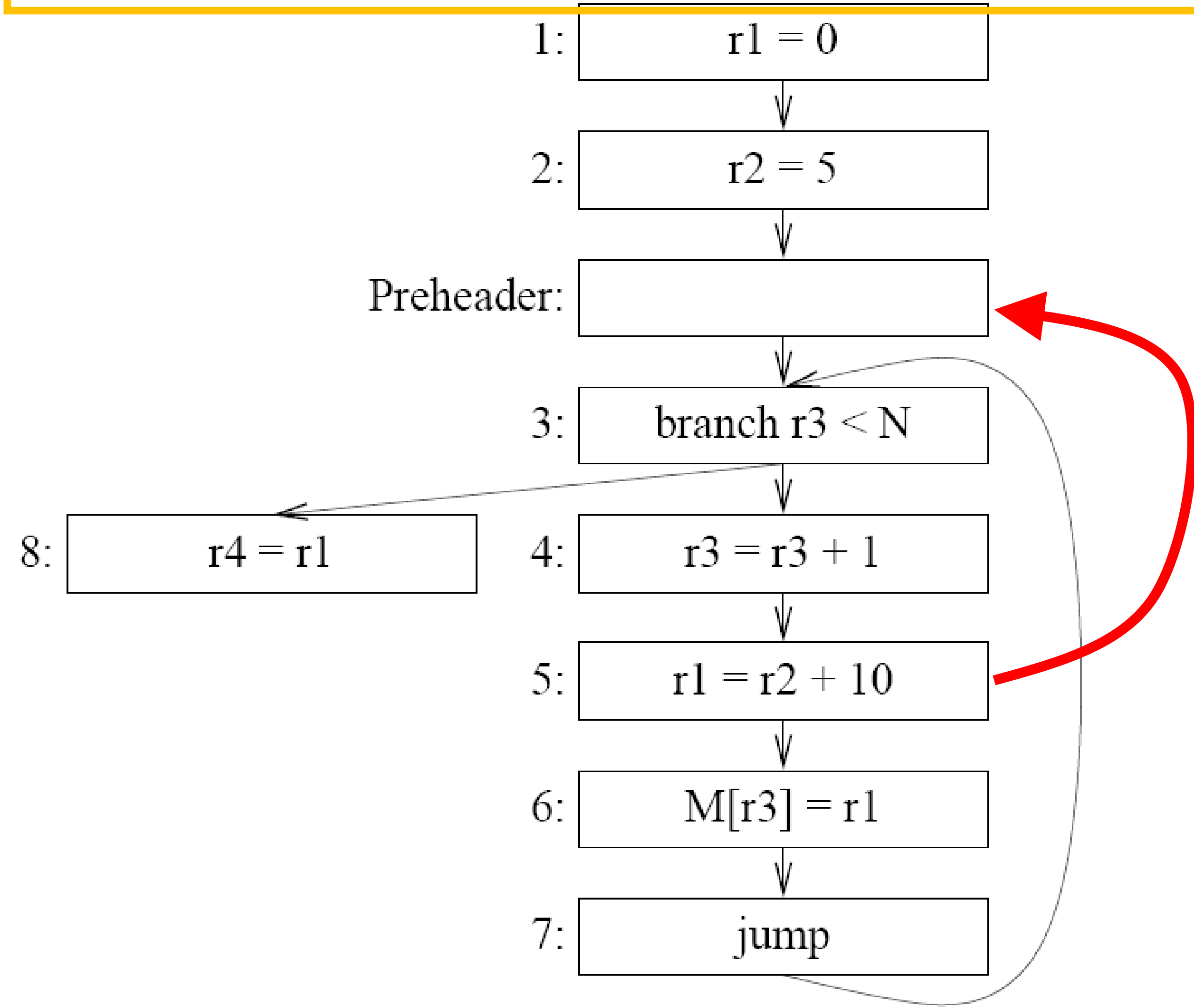


LICM: Constraint 1

d: `t = a op b`

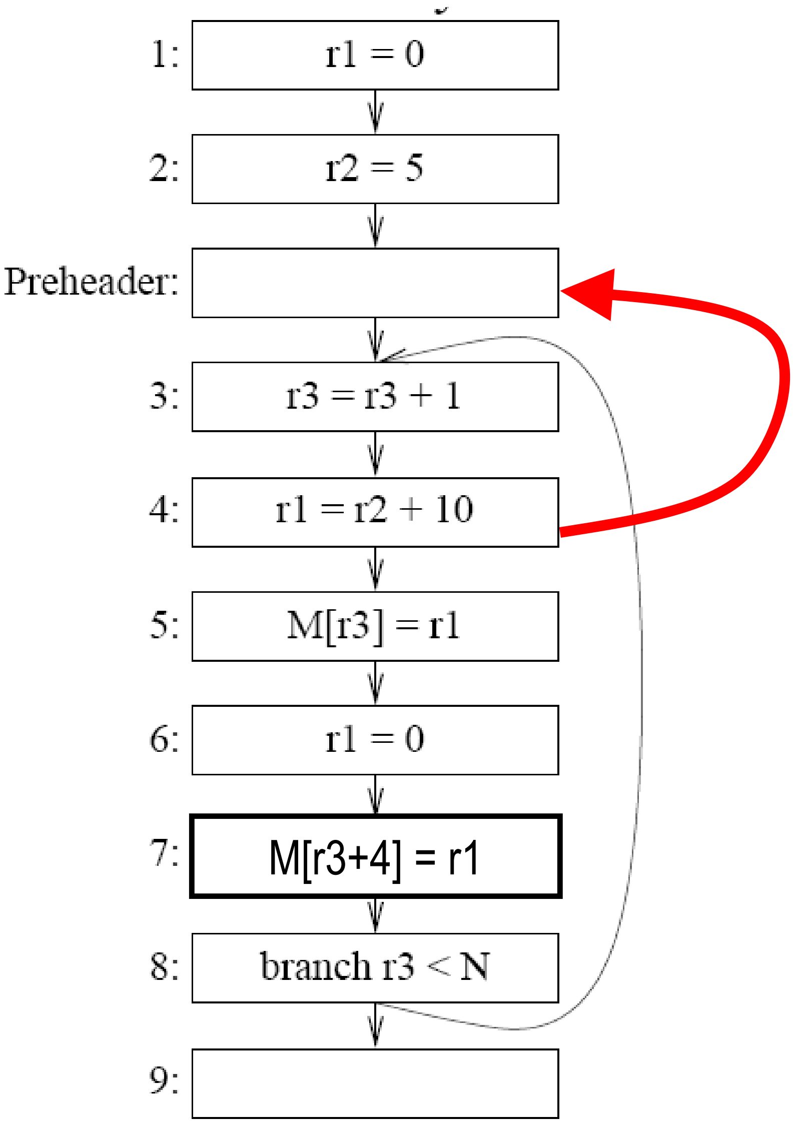
d must dominate all loop exit nodes where `t` is live out.

Constraint 1



Moving 5: r1 = r2+10 into preheader is illegal: final value of **r4** incorrect.

LICM: motivating constraint 2



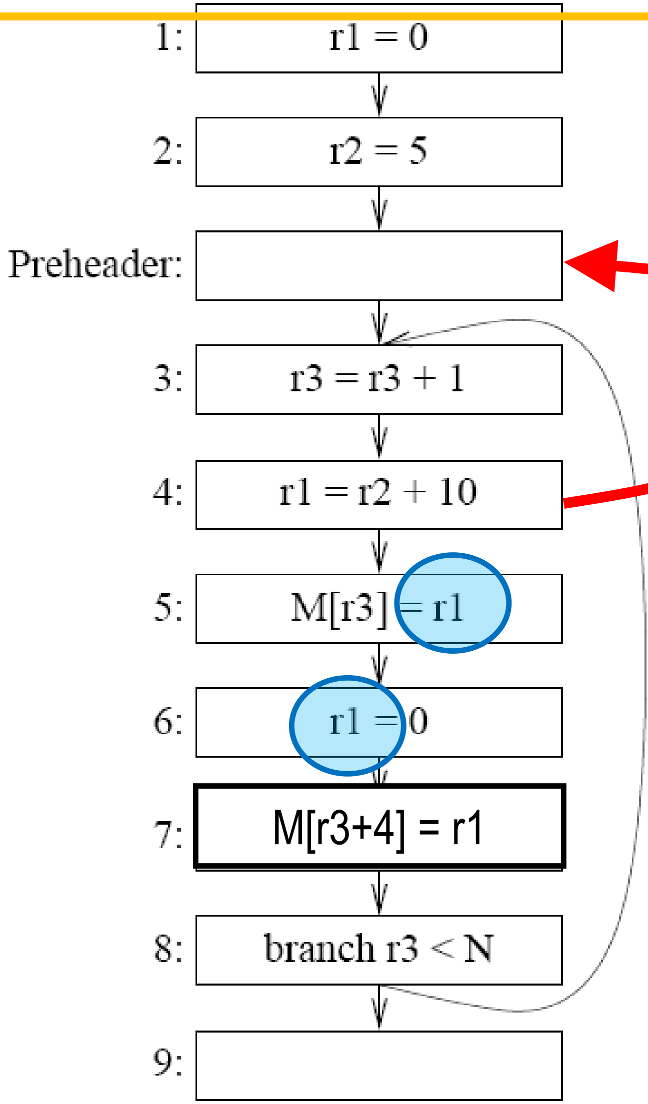
What could go wrong?

LICM: Constraint 2

d: `t = a op b`

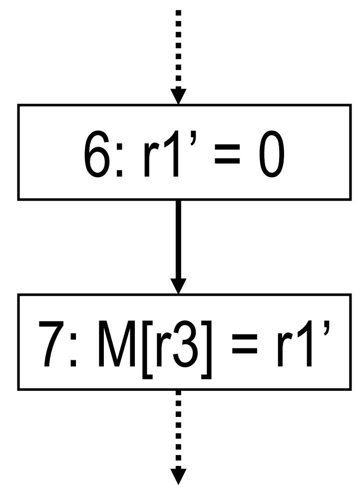
there must be only one definition of `t` inside loop.

Constraint 2



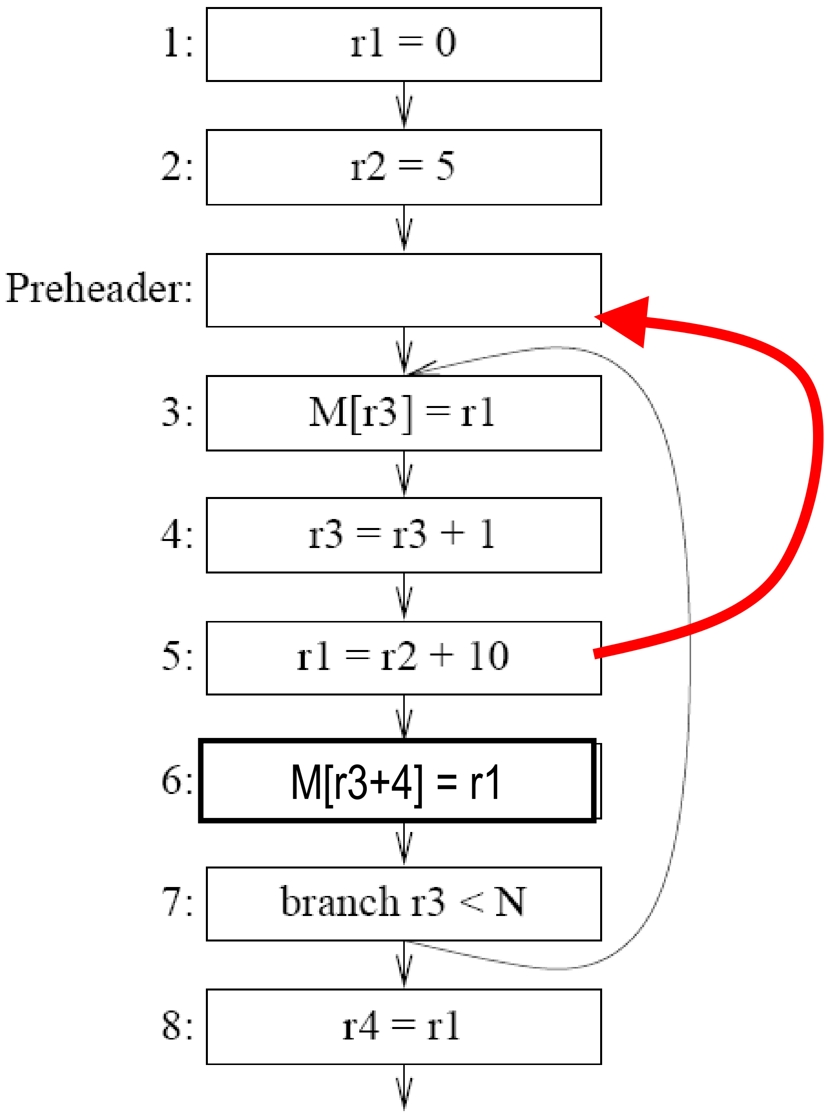
Moving $\boxed{\text{4: r1 = r2+10}}$ into preheader is illegal: second (and later) iteration would use incorrect value of **r1** in instruction **5**.

Possible solution: make variable names distinct:



Principled approach: SSA

LICM: motivating constraint 3



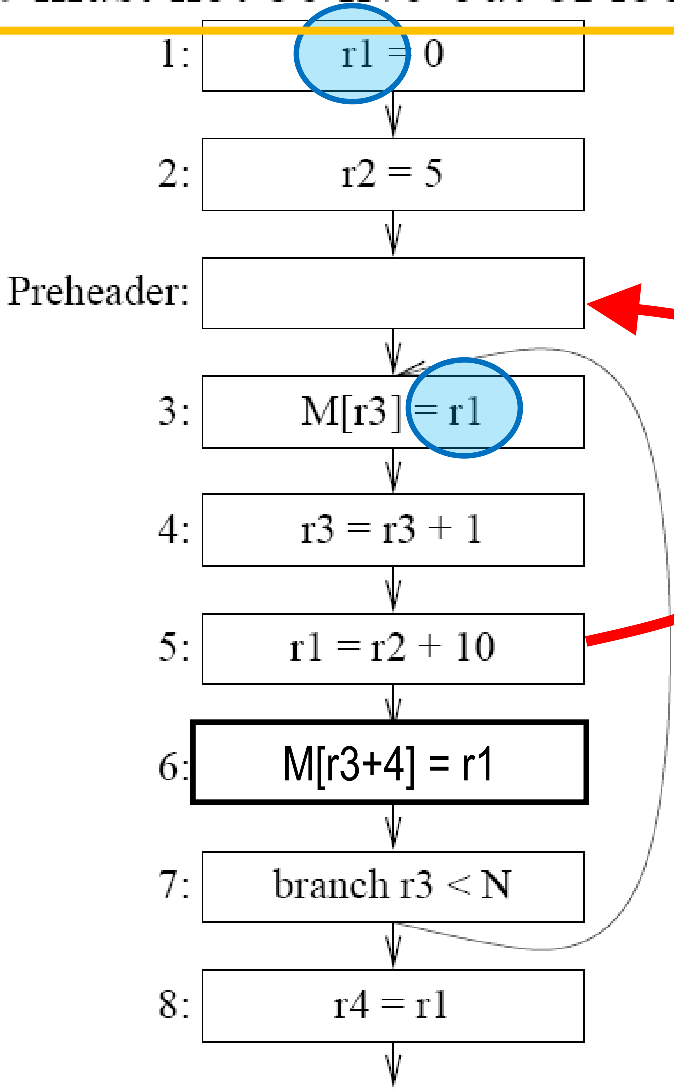
What could go wrong?

LICM: Constraint 3

d: `t = a op b`

t must not be live-out of loop preheader node (live-in to loop)

Constraint 3



Moving $\boxed{\text{5: r1 = r2+10}}$ into preheader is illegal: initial iteration would read incorrect value from **r1**

(SSA's variable renaming will get this right, using a trick…)

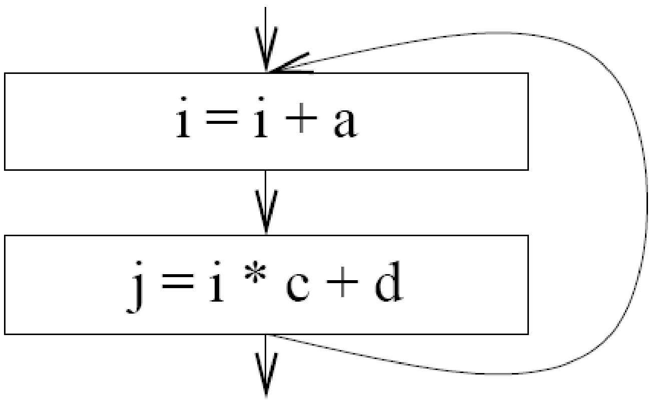
LICM

Algorithm for code motion:

- Examine invariant statements of L in same order in which they were marked.
- If invariant statement s satisfies three criteria for code motion, remove s from L, and insert into preheader node of L.

Induction Variables

Variable `i` in loop L is called induction variable of L if each time `i` changes value in L, it is incremented/decremented by loop-invariant value.



Assume `a`, `c` loop-invariant.

- `i` is an induction variable
- `j` is an induction variable
  - `j = i * c` is equivalent to `j = j + a * c`
  - compute `e = a * c` outside loop: `j = j + e` $\Rightarrow$ strength reduction
  - may not need to use `i` in loop $\Rightarrow$ induction variable elimination

j0 = i0 * c

i1 = i0 + a

j1 = i1 * c = (i0+a) * c = (i0 * c) + (a * c) = **j0 + a * c**

Induction Variable Detection

Scan loop L for two classes of induction variables:

- *basic* induction variables - variables (`i`) whose only definitions within L are of the form `i = i + c` or `i = i - c`, `c` is loop invariant.
- *derived* induction variables - variables (`j`) defined only once within L, whose value is linear function of some basic induction variable L.

Associate triple `(i, a, b)` with each induction variable `j`

- `i` is basic induction variable; `a` and `b` are loop invariant.
- value of `j` at point of definition is `a + b * i`
- `j` belongs to the family of `i`

Induction Variable Detection: Algorithm

Algorithm for induction variable detection:

- Scan statements of L for basic induction variables `i`
  - for each `i`, associate triple `(i, 0, 1)`
  - `i` belongs to its own family.

$1 \cdot i + 0 = i$

Induction Variable Detection: Algorithm

Algorithm for induction variable detection:

- Scan statements of L for basic induction variables `i`
  - for each `i`, associate triple `(i, 0, 1)` $1 \cdot i + 0 = i$
  - `i` belongs to its own family.
- Scan statements of L for derived induction variables `k`:
  1. there must be single assignment to `k` within L of the form `k = j * c` or `k = j + d`, `j` is an induction variable; `c`, `d` loop-invariant, and
  2.

Induction Variable Detection: Algorithm

Algorithm for induction variable detection:

- Scan statements of L for basic induction variables `i`
  - for each `i`, associate triple `(i, 0, 1)`
  - `i` belongs to its own family.

$1 \cdot i + 0 = i$

- Scan statements of L for derived induction variables `k`:
  1. there must be single assignment to `k` within L of the form `k = j * c` or `k = j + d`, `j` is an induction variable; `c`, `d` loop-invariant, and
  2. if `j` is a derived induction variable belonging to the family of i, then:
     - the only definition of `j` that reaches `k` must be one in L, and
     - no definition of `i` must occur on any path between definition of `j` and definition of `k`

Induction Variable Detection: Algorithm

Algorithm for induction variable detection:

- Scan statements of L for basic induction variables `i`
  - for each `i`, associate triple `(i, 0, 1)`
  - `i` belongs to its own family.

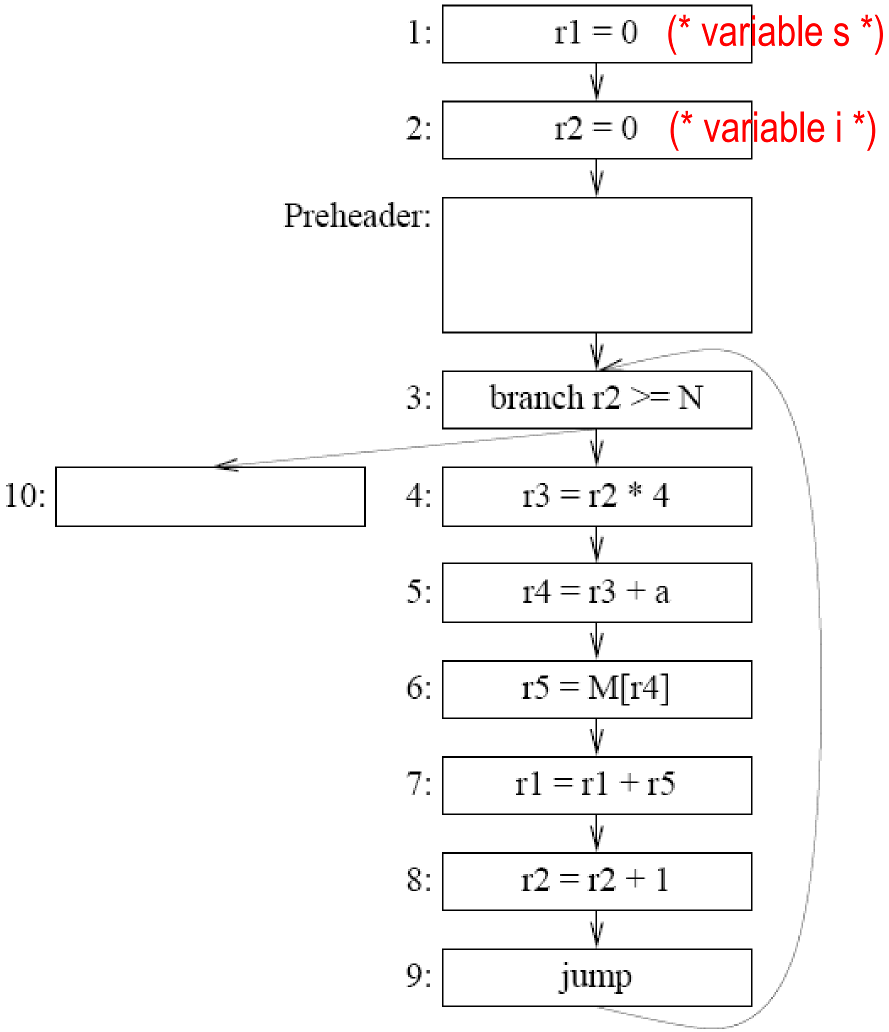
$1 \cdot i + 0 = i$

- Scan statements of L for derived induction variables `k`:
  1. there must be single assignment to `k` within L of the form `k = j * c` or `k = j + d`, `j` is an induction variable; `c`, `d` loop-invariant, and
  2. if `j` is a derived induction variable belonging to the family of i, then:
     - the only definition of `j` that reaches `k` must be one in L, and
     - no definition of `i` must occur on any path between definition of `j` and definition of `k`
- Assume `j` associated with triple `(i, a, b)`: `j = a + b * i` at point of definition.
- Can determine triple for `k` based on triple for `j` and instruction defining `k`:
  - `k = j * c` → `(i, a*c, b*c)`
  - `k = j + d` → `(i, a + d, b)`

In general: k = j*c + d → (i, a*c+d, b*c), but there's usually no instruction form k = j*c + d…

Induction Variable Detection: Example

```
s = 0;
for(i = 0; i < N; i++)
  s += a[i];
```

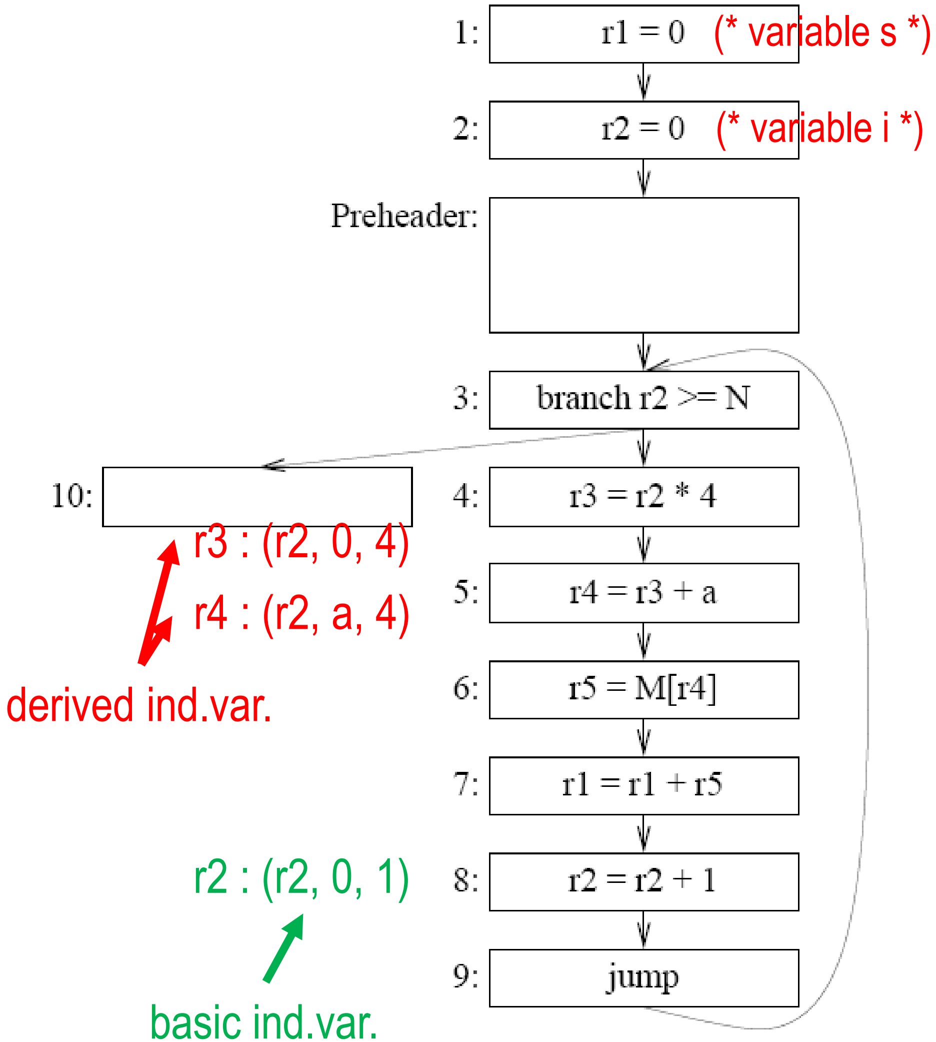


basic induction variable(s)?

derived induction variables?

Induction Variable Detection: Example

```
s = 0;
for(i = 0; i < N; i++)
  s += a[i];
```



Strength Reduction: replace by cheaper instruction

1. For each derived induction variable `j` with triple `(i, a, b)`, create new `j'`.
   - all derived induction variables with same triple `(i, a, b)` may share `j'`
2. After each definition of `i` in L, `i = i + c`, insert statement:

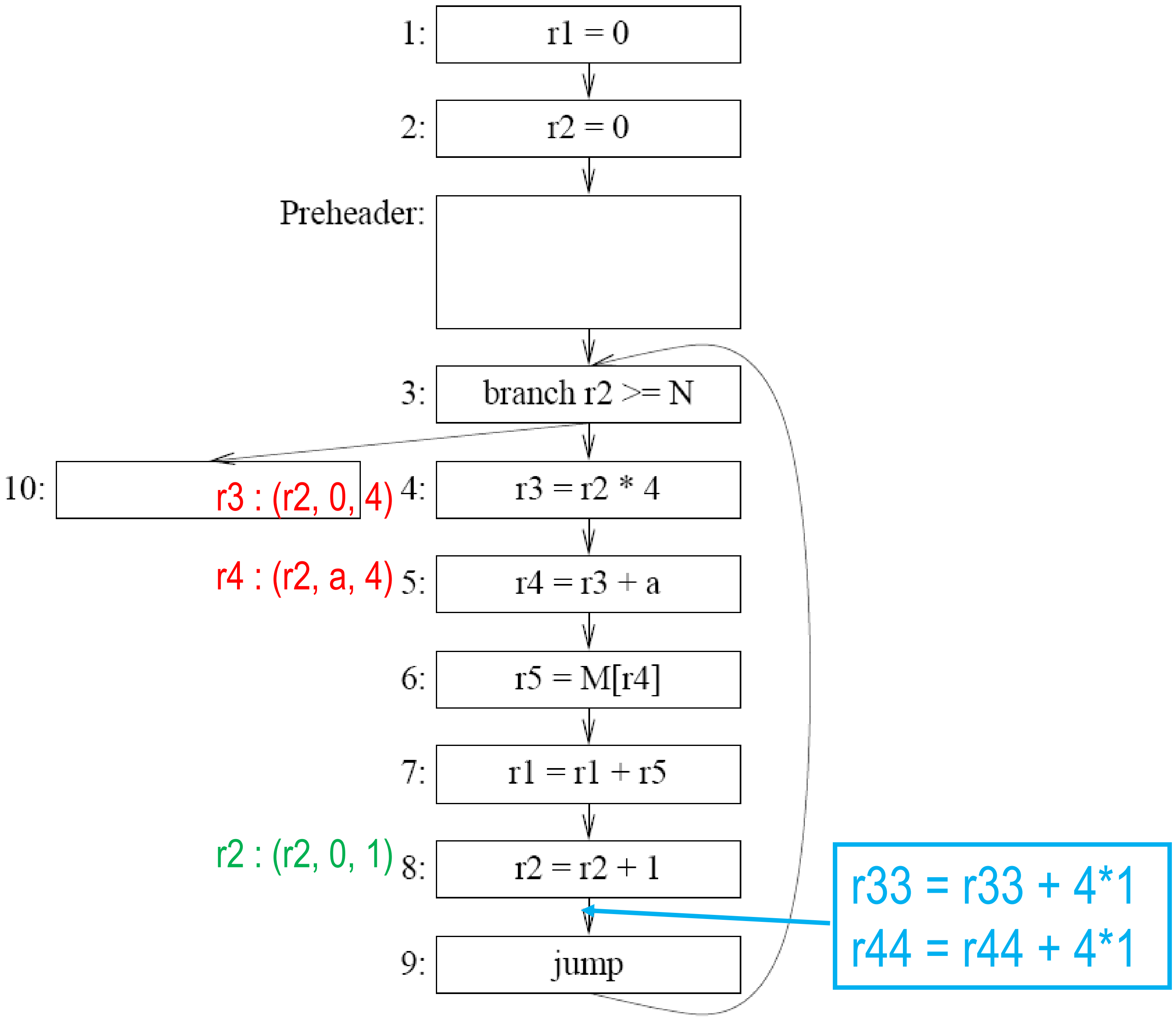
```
j' = j' + b * c
```

   - `b * c` is loop-invariant and may be computed in preheader or during compile time.
3. Replace unique assignment to `j` with `j = j'`.
4. Initialize `j'` at end of preheader node:

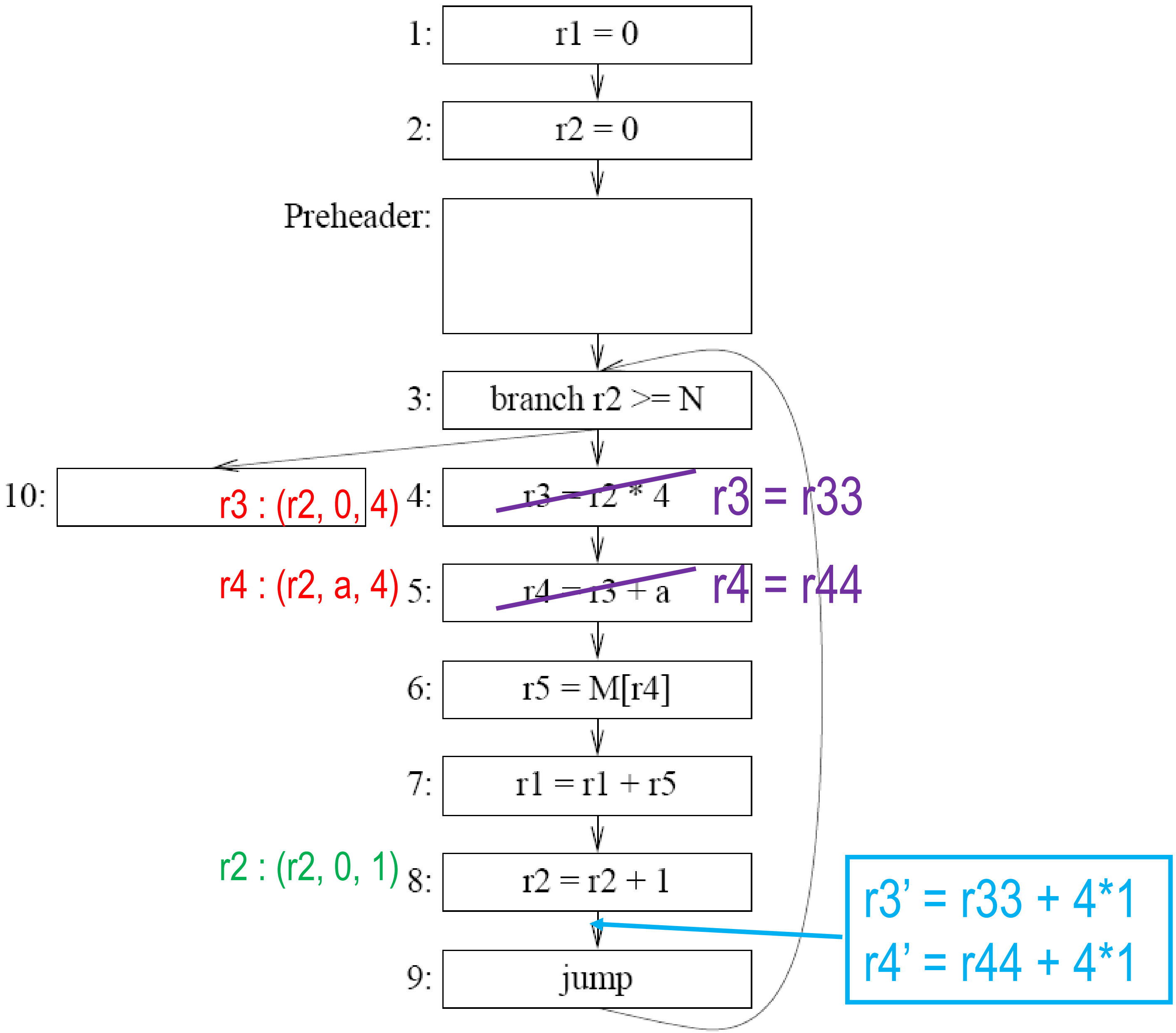
```
j' = b * i
j' = j' + a
```

- Strength reduction still requires multiplication, but multiplication now performed outside loop.
- `j'` also has triple `(i, a, b)`

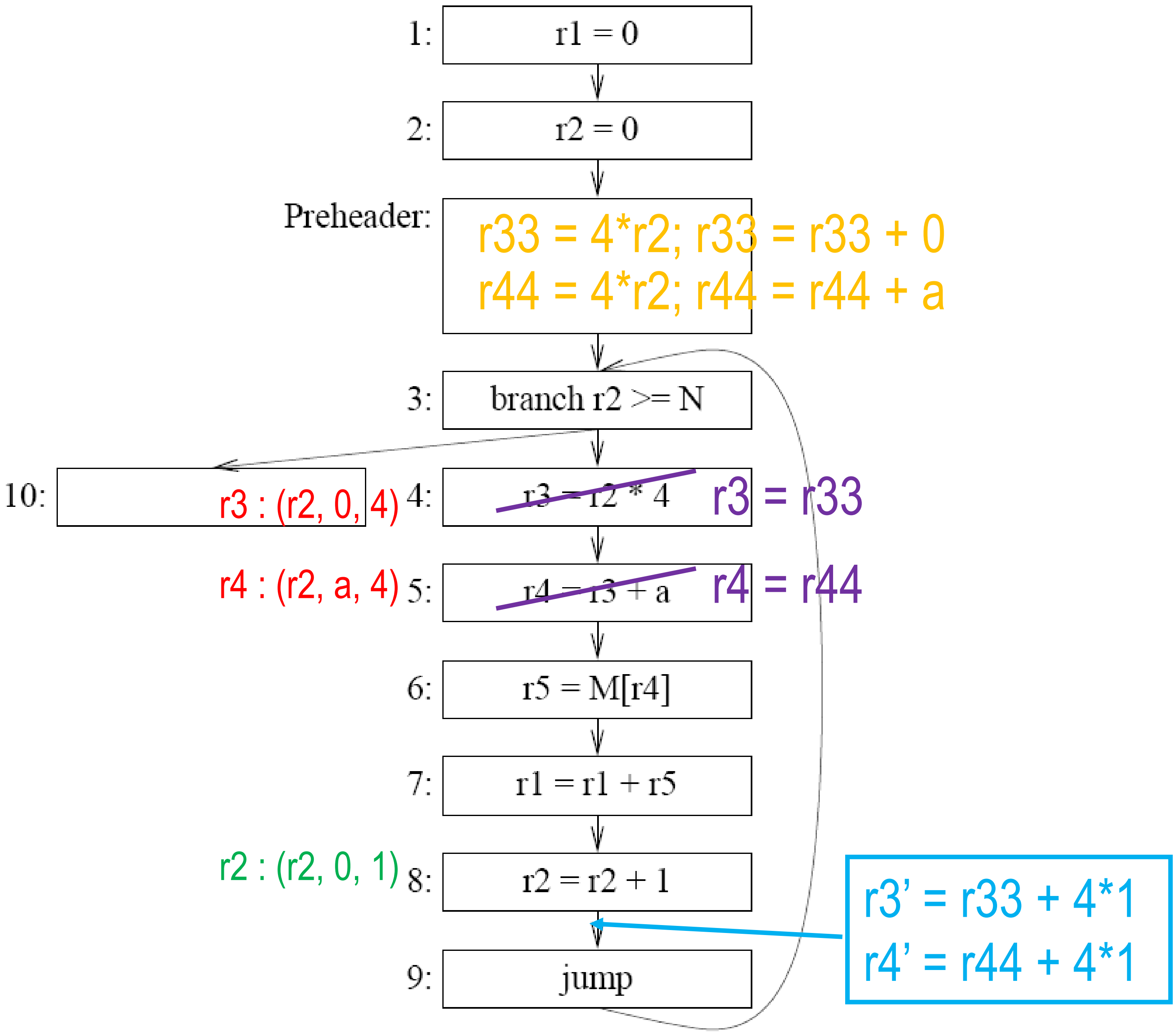
Strength Reduction Example



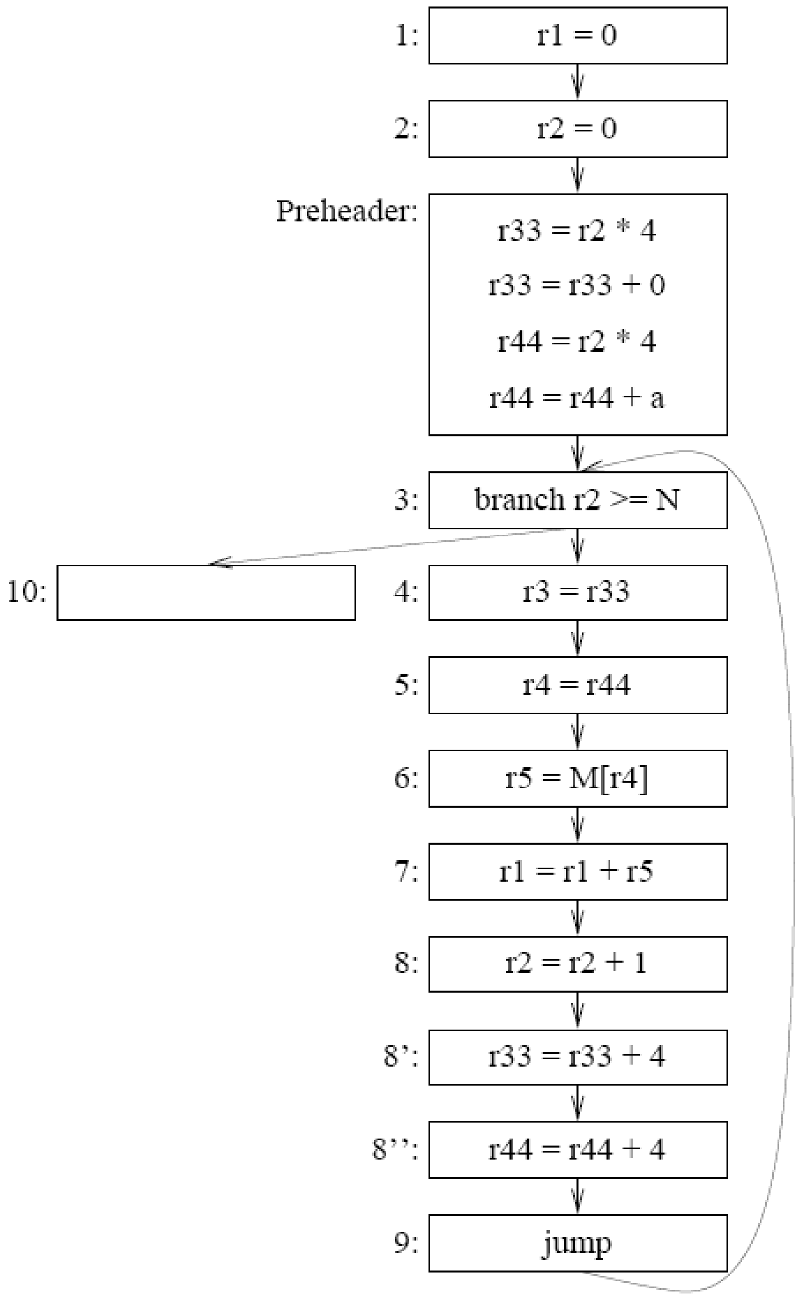
Strength Reduction Example



Strength Reduction Example



Strength Reduction Example



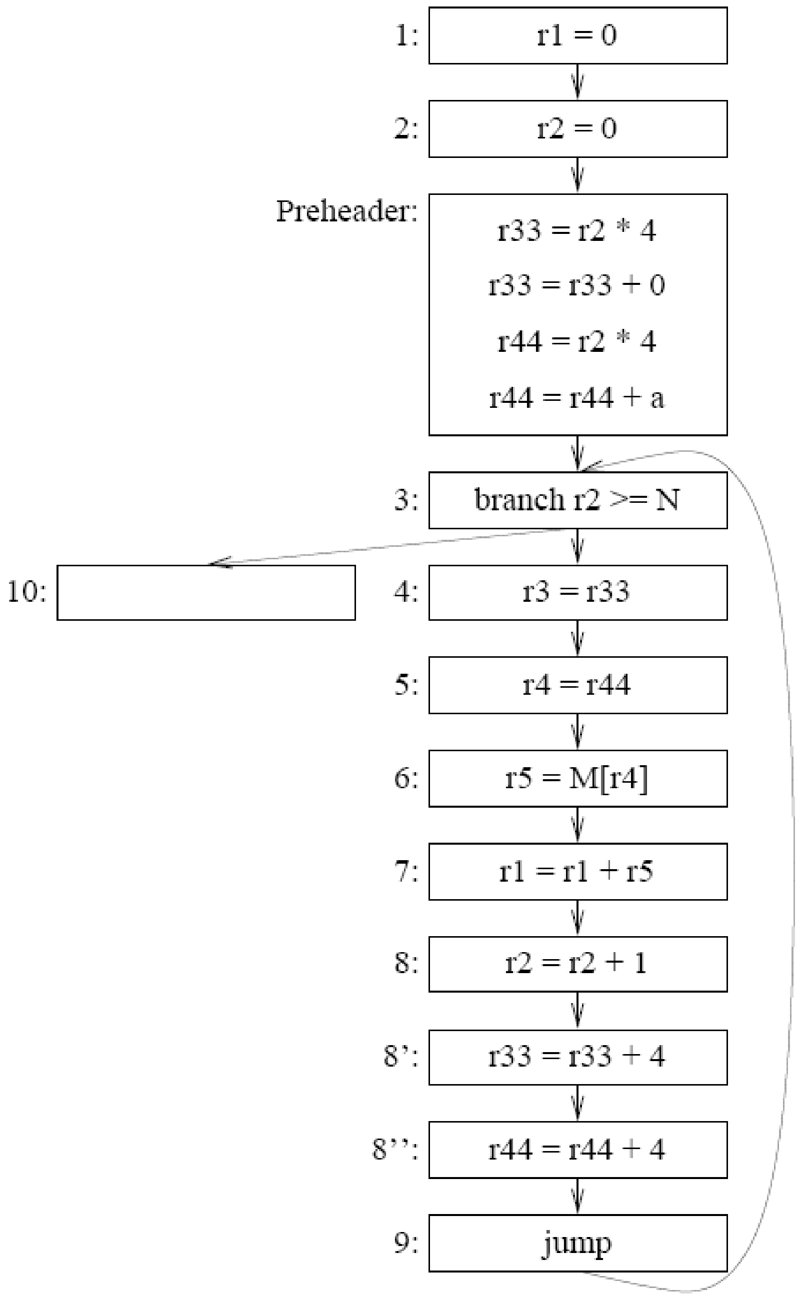
Strength reduction introduces more opportunities for code optimization . . .

Induction Variable Elimination

After strength reduction has been performed:

- some induction variables are only used in comparisons with loop-invariant values.
- some induction variables are *useless*
  - dead on all loop exits, used only in definition of itself.
  - dead code elimination will not remove useless induction variables.

Induction Variable Elimination Example

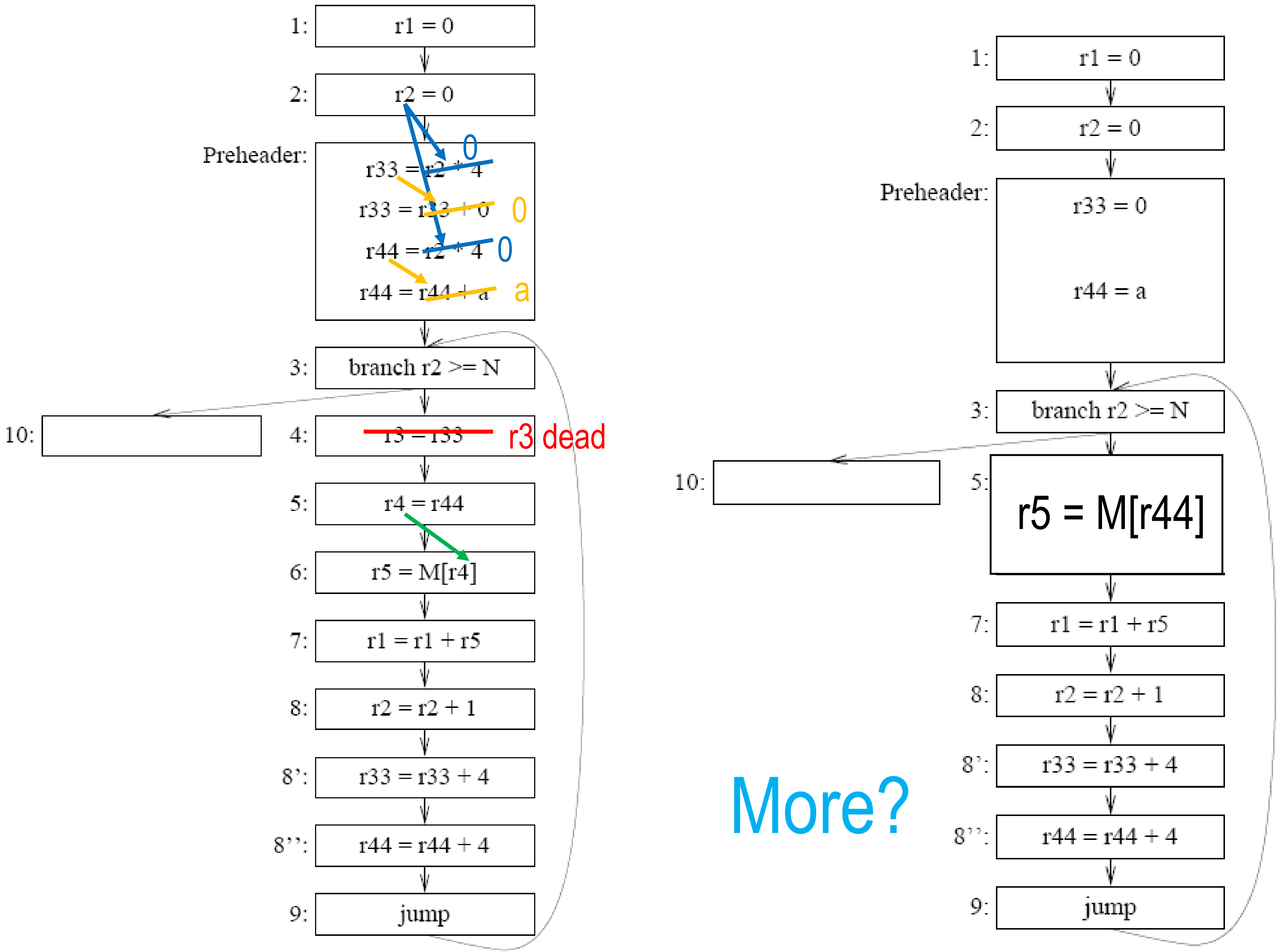


Any dead assignments?

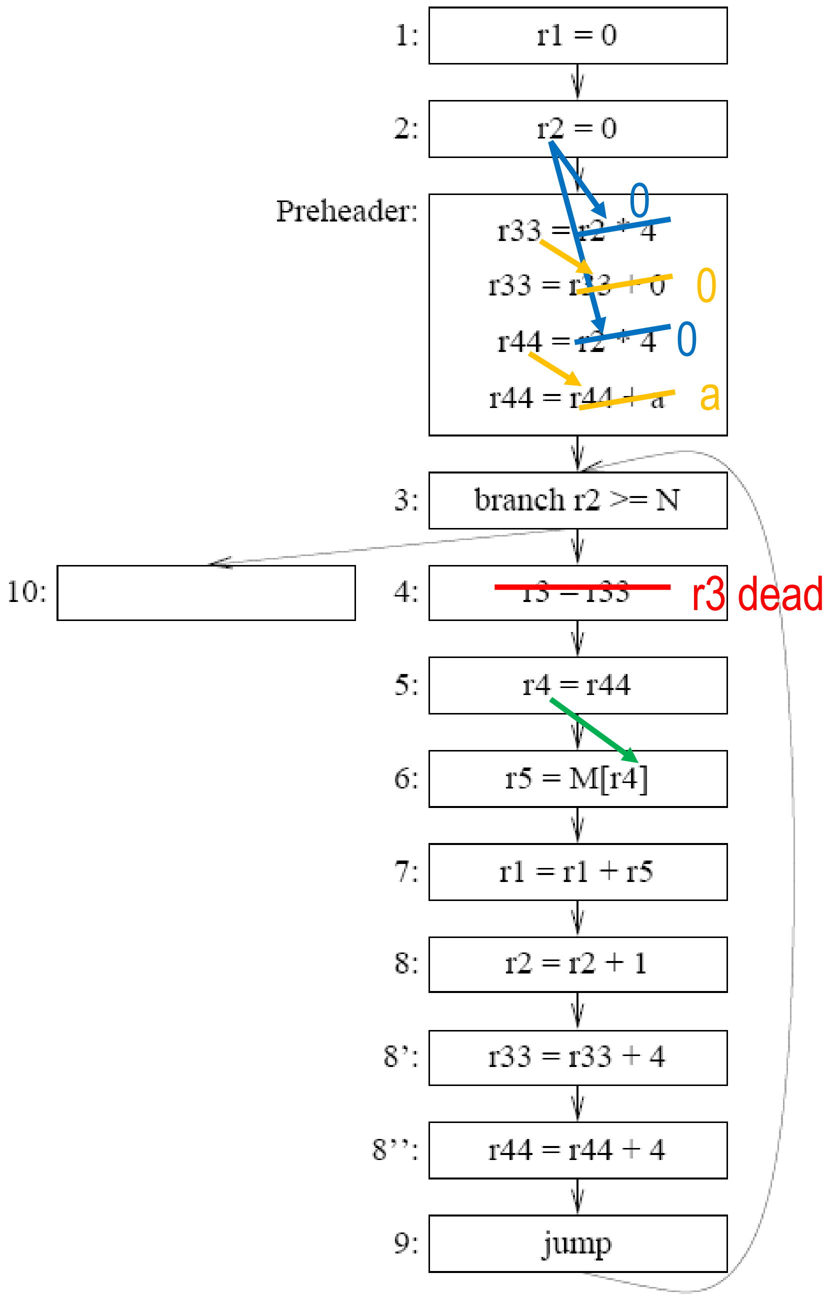
Useless variables?

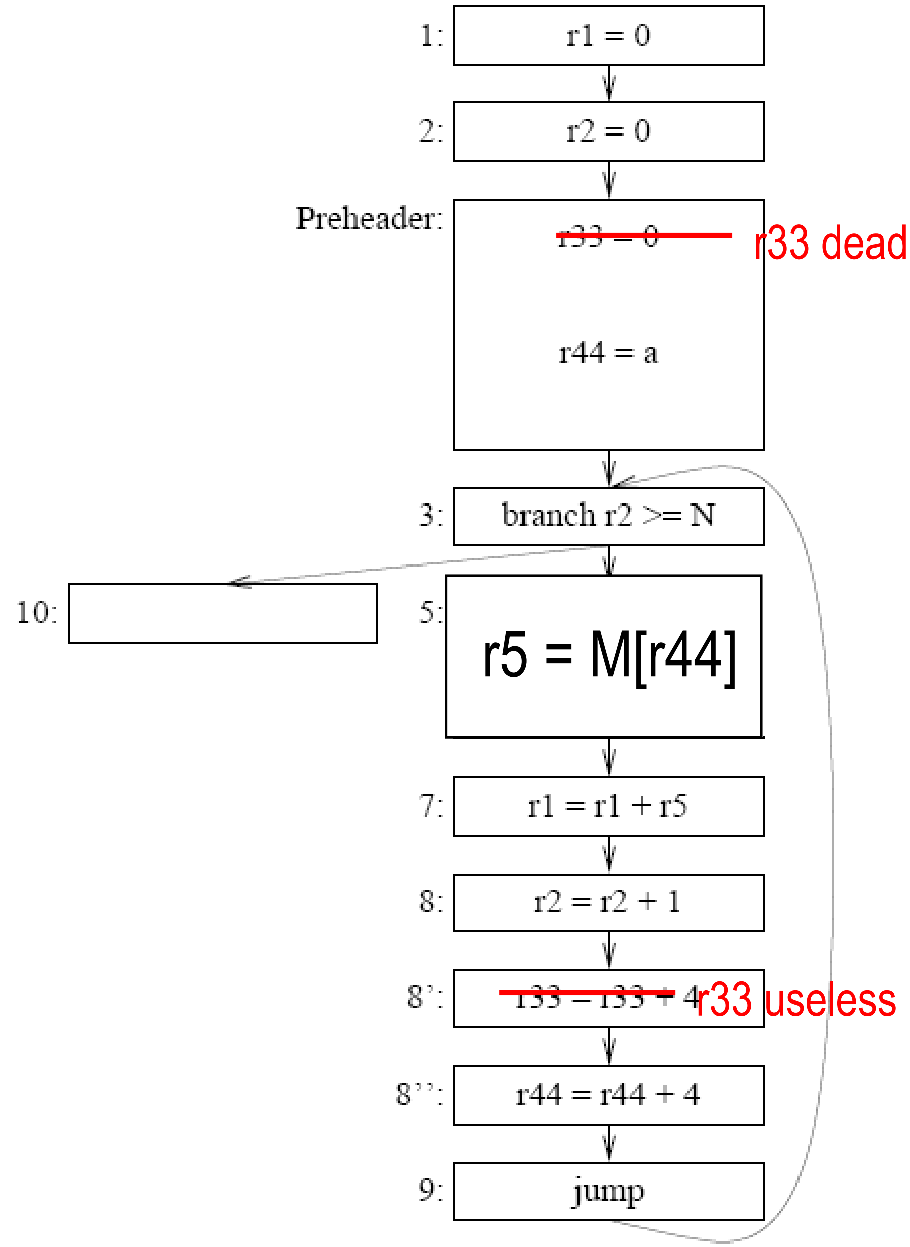
Copy propagation?

Induction Variable Elimination Example



Induction Variable Elimination Example

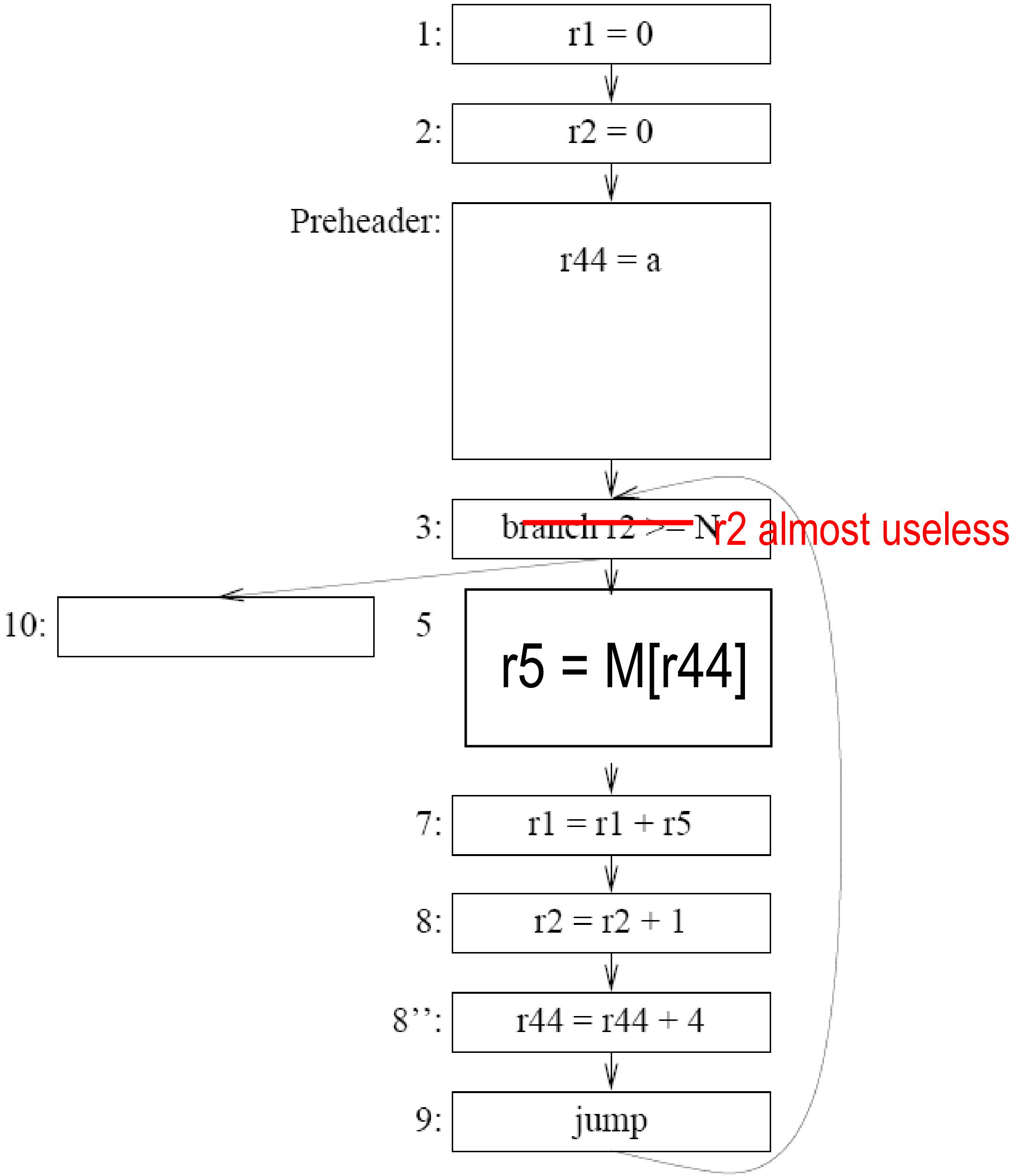




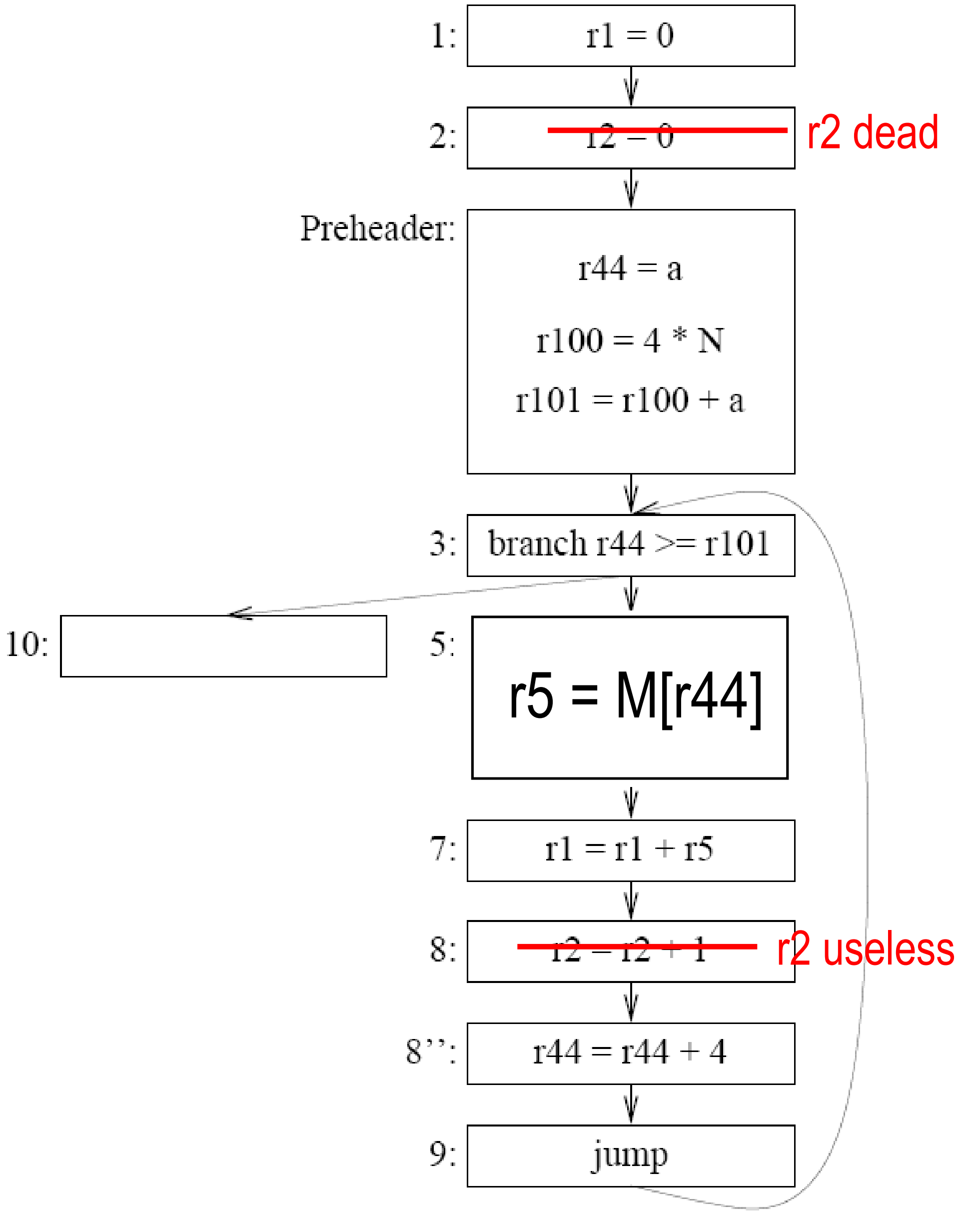
Induction Variable Elimination

- Variable `k` is *almost useless* if it is only used in comparisons with loop-invariant values, and there exists another induction variable `t` in the same family as `k` that is not useless.
- Replace `k` in comparison with `t`
  $\rightarrow$ `k` is useless

Induction Variable Elimination: Example



Induction Variable Elimination: Example



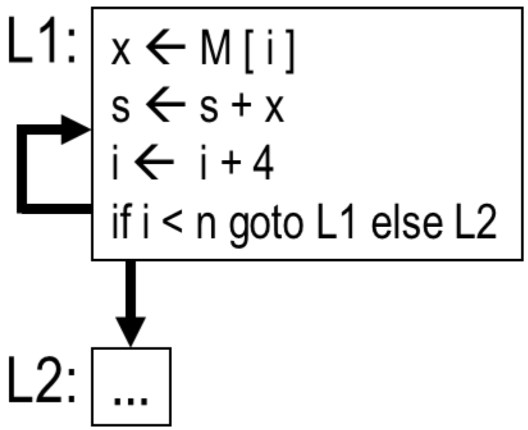
No more optimizations for now.

Loop unrolling: Example

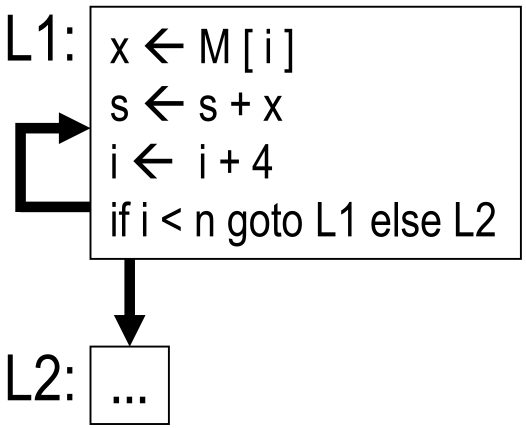
Idea: combine several iterations of a loop

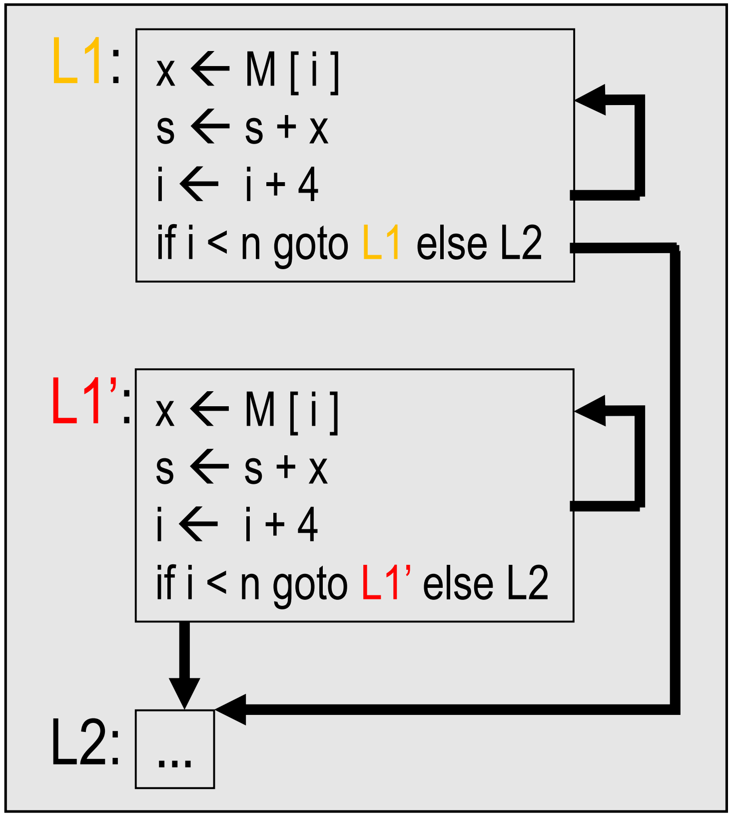
- # iterations static constant: can unroll fully to straight-line code, eliminating comparison/jump operation
- # iterations fixed (ie loop bound does not change inside the body):
  - reduces #iterations/conditional jumps/induction variable increments
  - occasionally beneficial for parallelization, scheduling

Running example:



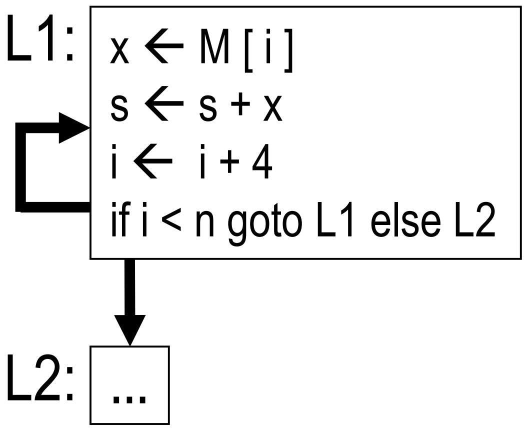
Naïve loop unrolling: simple example

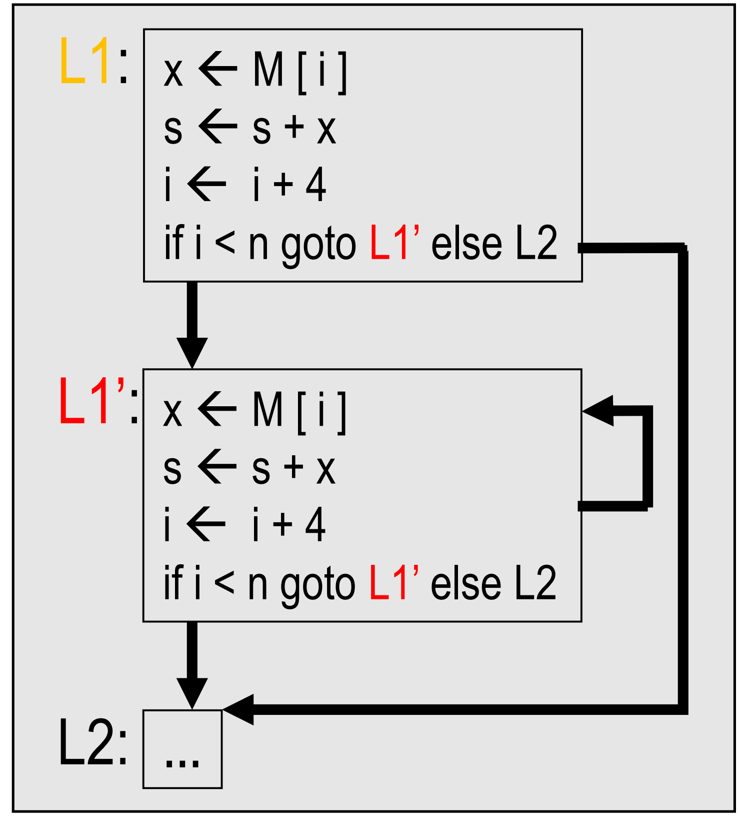




1. Copy loop to make L' with header h' and back edges $s_i' \rightarrow h'$

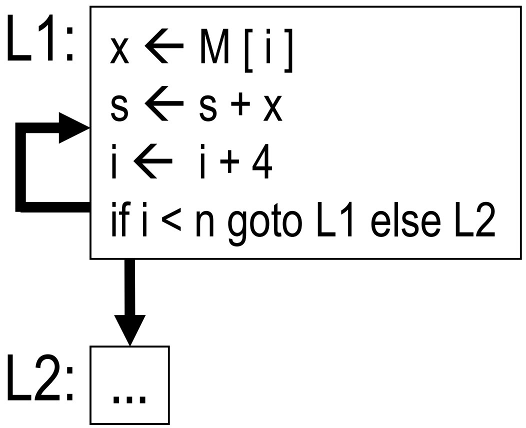
Naïve loop unrolling: simple example

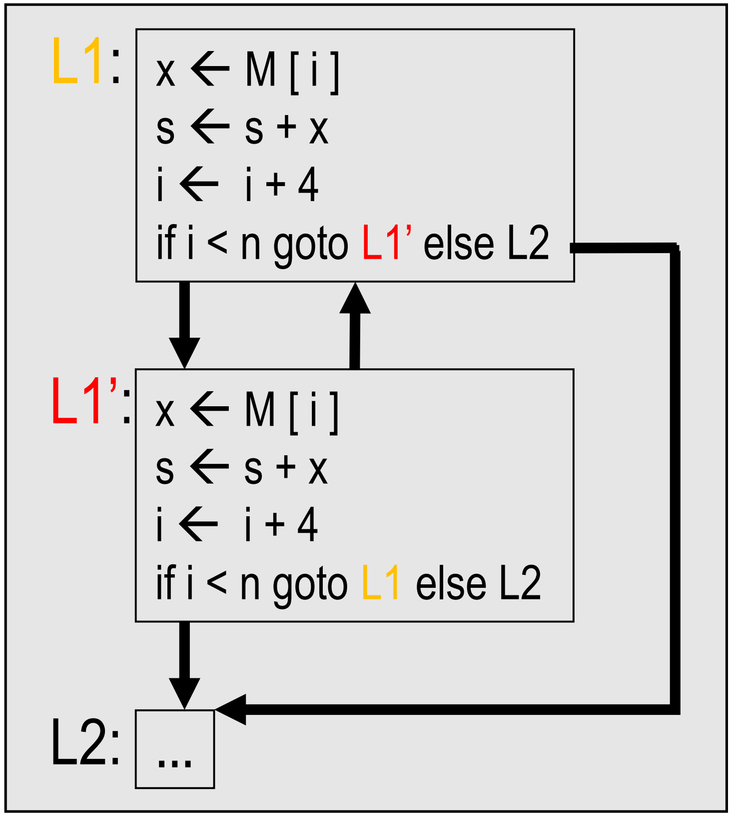




1. Copy loop to make L' with header h' and back edges $s_i' \rightarrow h'$
2. Change back edges in L from $s_i \rightarrow h$ to $s_i \rightarrow h'$

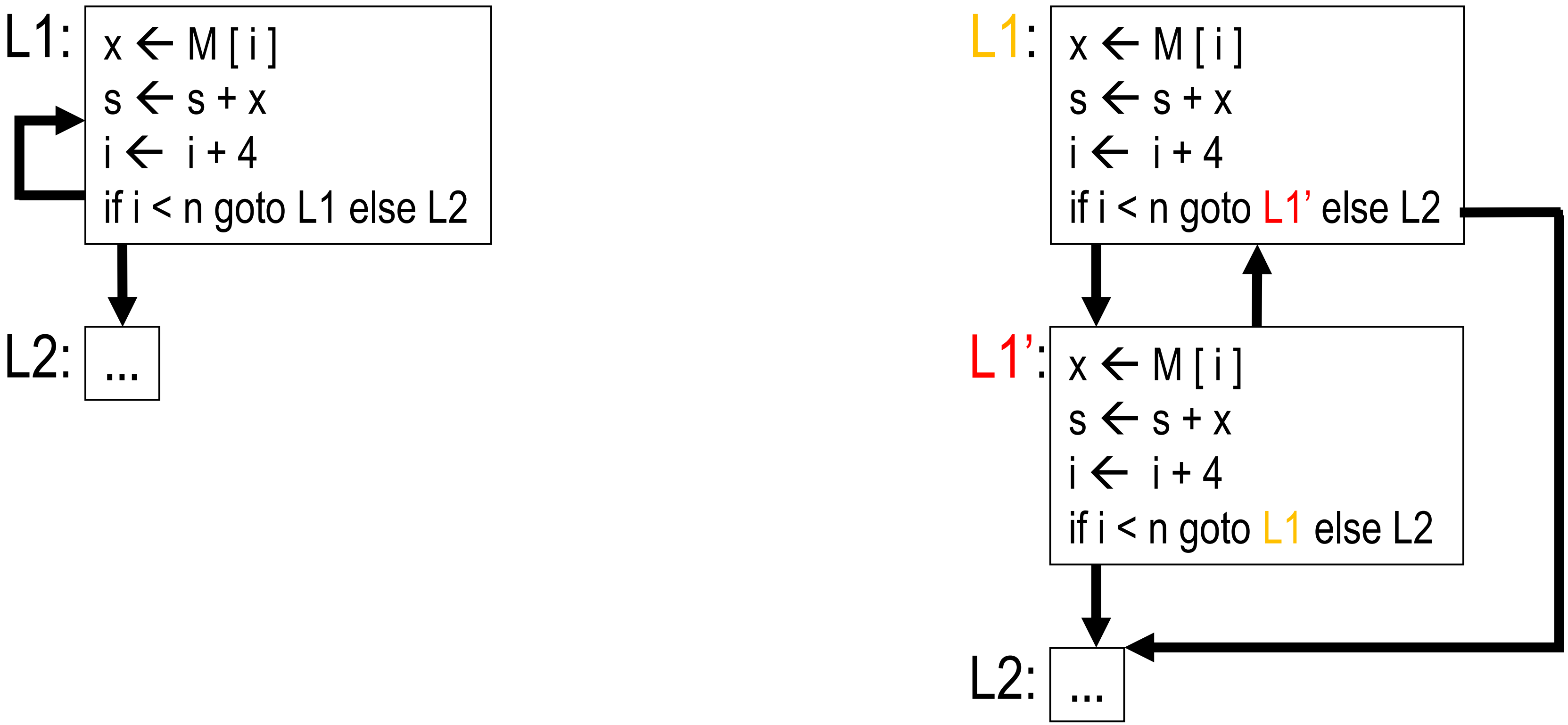
Naïve loop unrolling: simple example





1. Copy loop to make L’ with header h’ and back edges $s_i' \rightarrow h'$
2. Change back edges in L from $s_i \rightarrow h$ to $s_i \rightarrow h'$
3. Change back edges in L’ from $s_i' \rightarrow h'$ to $s_i' \rightarrow h$

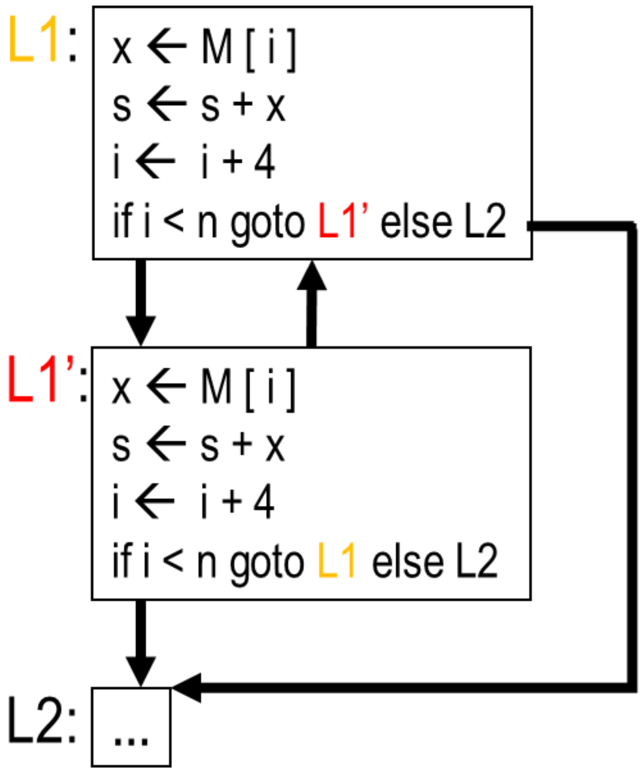
Naïve loop unrolling: simple example



1. Copy loop to make L' with header h' and back edges $s_i' \rightarrow h'$
2. Change back edges in L from $s_i \rightarrow h$ to $s_i \rightarrow h'$
3. Change back edges in L' from $s_i' \rightarrow h'$ to $s_i' \rightarrow h$

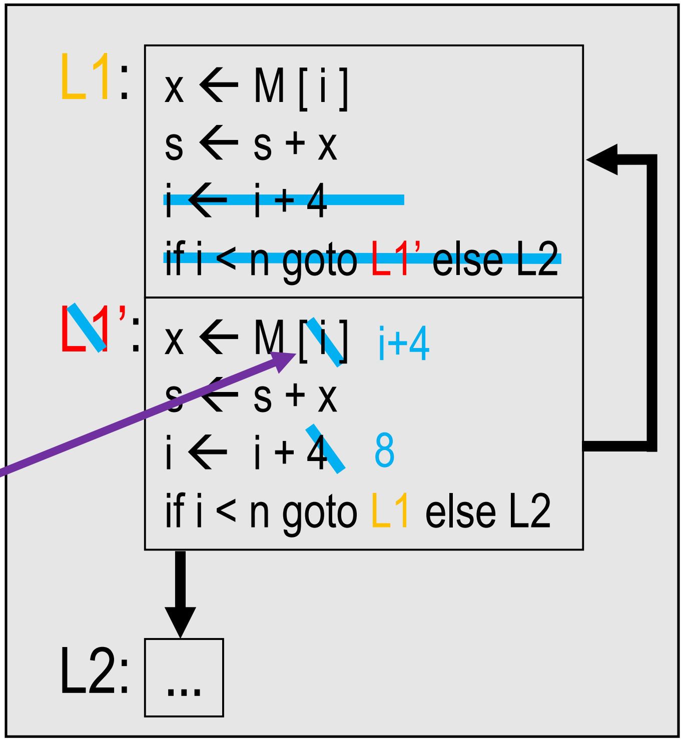
But: little optimization – still 2 increments and 2 conditional jumps…

Loop unrolling: “optimistic” merging of bodies



Observe: only one back edge, and both increments to the induction variable i dominate this back edge.

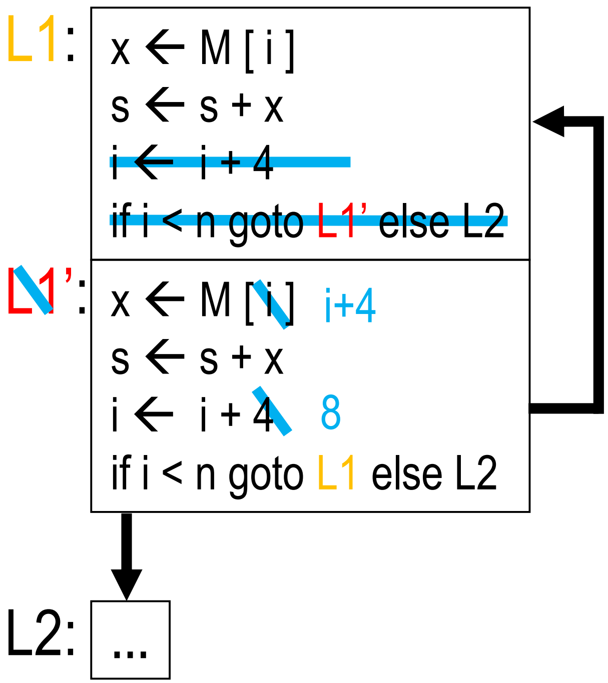
Naïve merging of L1 and L1’:

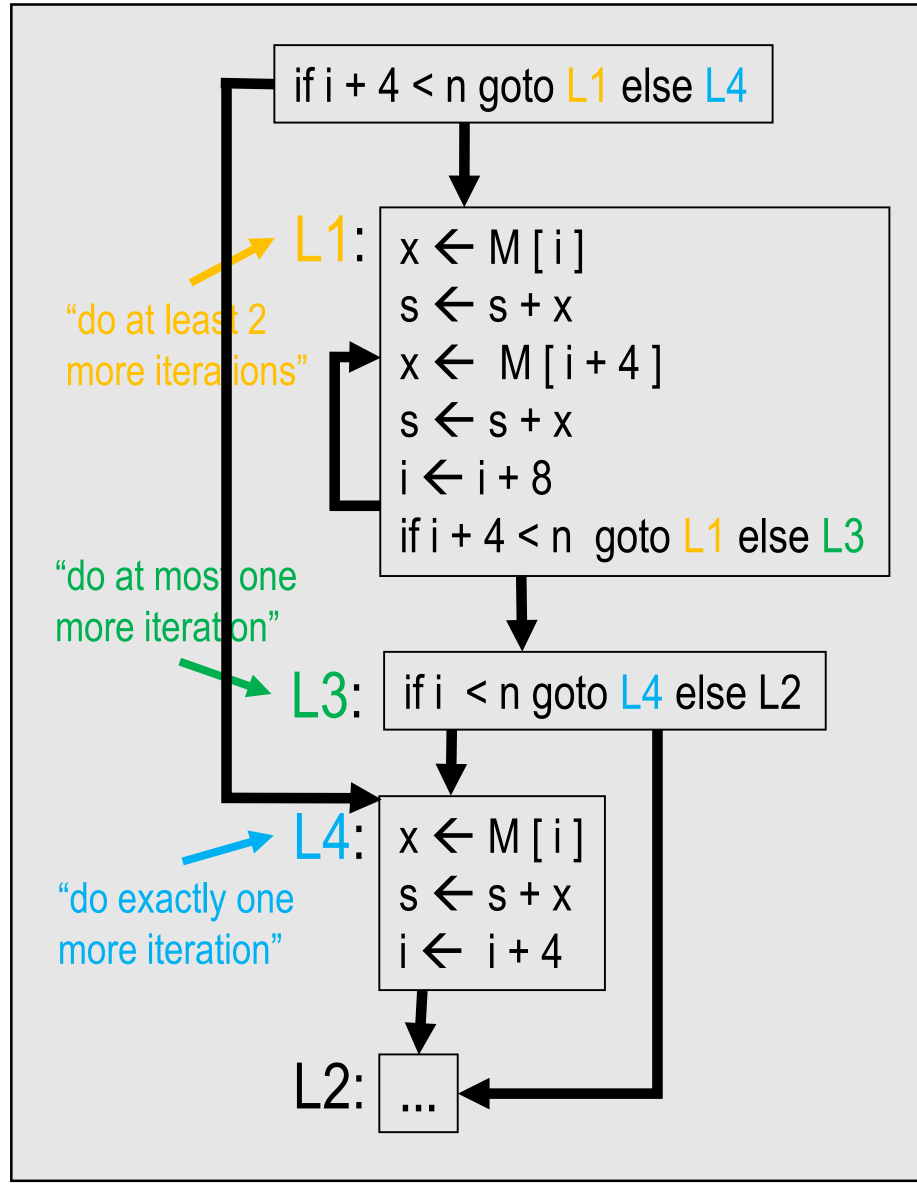


i+4 still needs to be computed…

But: only correct if original loop performed even number of iterations!

Loop unrolling: correcting optimistic merge

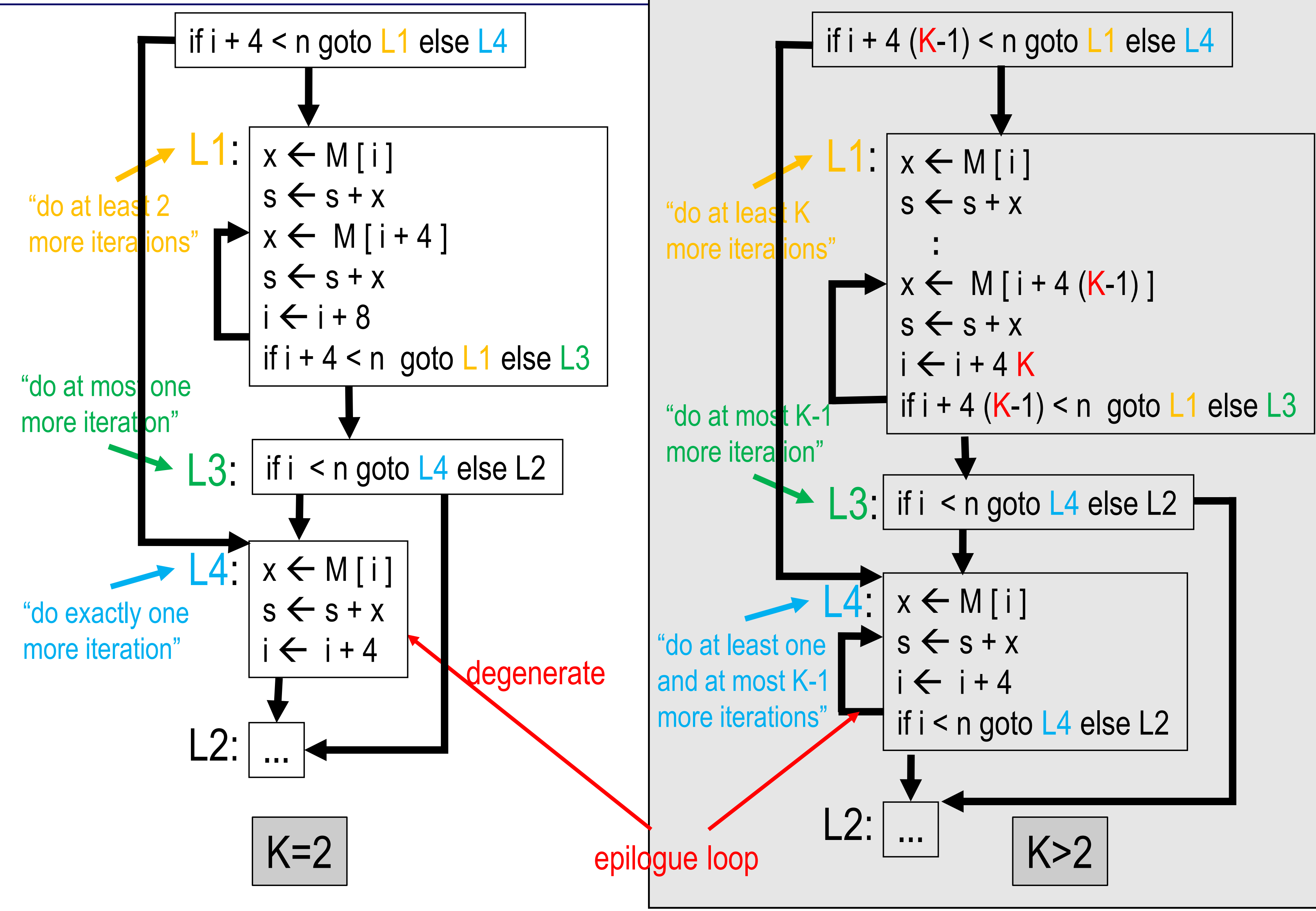




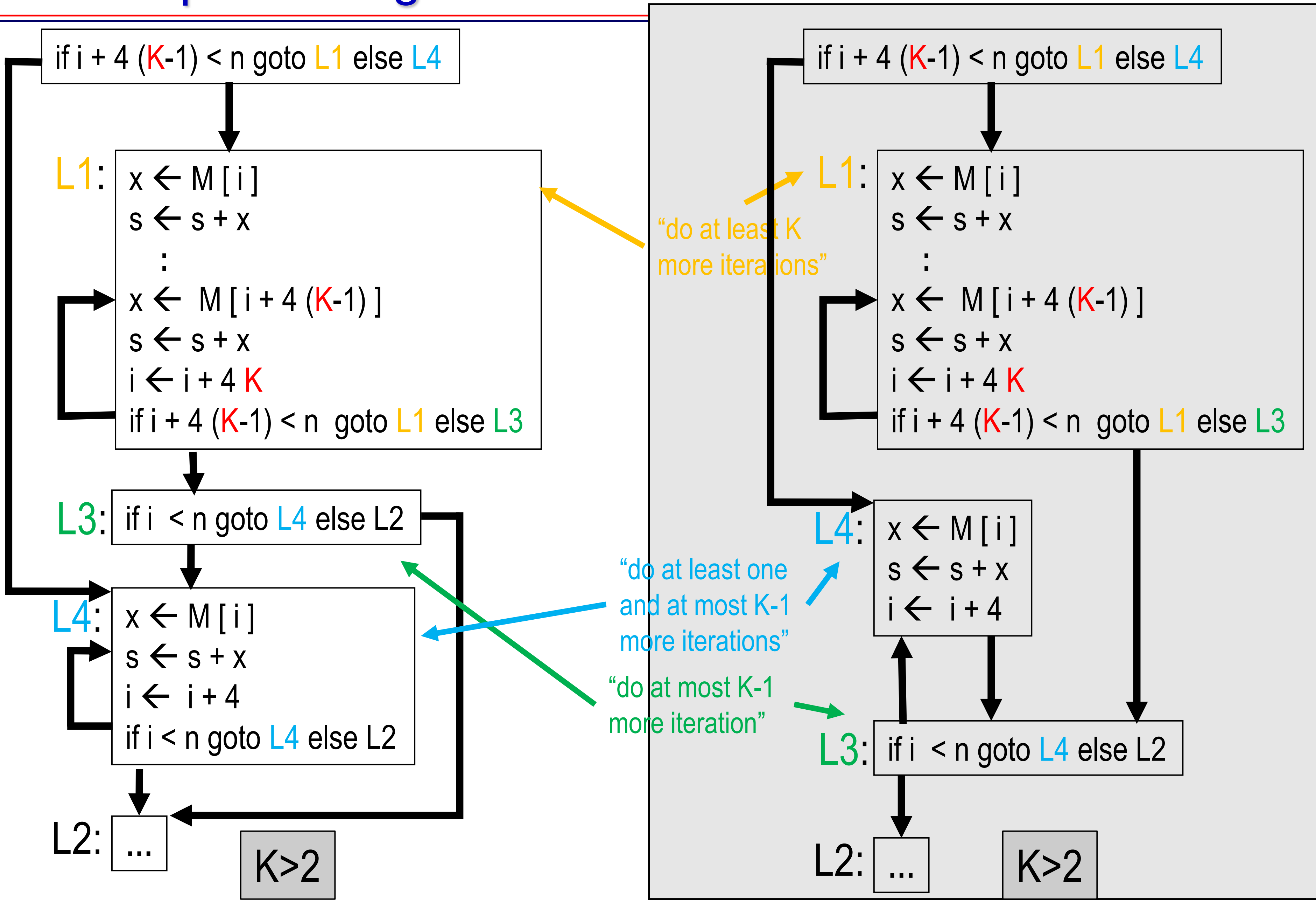
Execute remaining iteration (if necessary) in a new loop epilogue and adjust control flow !

Program different from Program 18.11(b) in MCIML, page 424!

Loop unrolling: unroll K iterations



Loop unrolling: unroll K iterations



Swapping order of blocks L3/L4 optimizes code size – at the price of irreducibility!