Loop Optimizations

- Large fraction of execution time is spent in loops.
- Effective loop optimization is extremely important.
- First step in loop optimization $\rightarrow$ find the loops.
- 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 Optimization

Loop invariant code motion

- An instruction is loop invariant if it computes the same value in each iteration.
- Invariant code may be hoisted outside the loop.

```
  ADDI   r1 = r0
  LOAD   r2 = M[FP + a]
  ADDI   r3 = r0 + 4

LOOP:
  MUL    r4 = r3 * r1
  ADD    r5 = r5 + r4
  LOAD   r6 = M[r3]

  ADDI   r1 = r1 + 1
  BRANCH r1 <= 10, LOOP
```

Loop Optimization

Loop invariant code motion

- An instruction is loop invariant if it computes the same value in each iteration.
- Invariant code may be hoisted outside the loop.

```
  ADDI   r1 = r0
  LOAD   r2 = M[FP + a]
  ADDI   r3 = r0 + 4
  LOAD   r6 = M[r3]

LOOP:
  MUL    r4 = r3 * r1
  ADD    r5 = r5 + r4

  ADDI   r1 = r1 + 1
  BRANCH r1 <= 10, LOOP
```

Loop Optimization

- **Induction variable analysis and elimination** - i is an induction variable if only definitions of i within loop increment/decrement i by loop-invariant value.
- **Strength reduction** - replace expensive instructions (like multiply) with cheaper ones (like add).

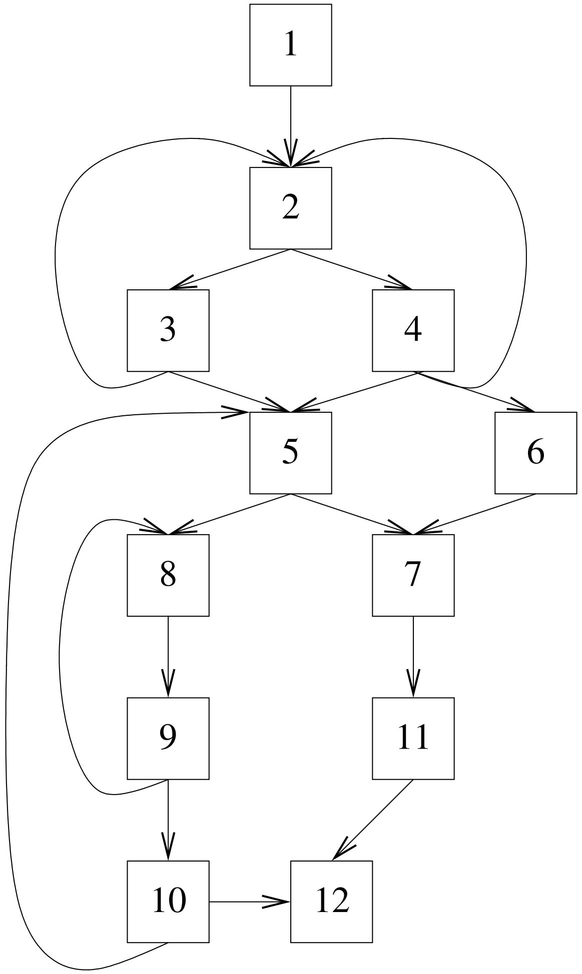
```
  ADDI   r1 = r0
  LOAD   r2 = M[FP + a]
  ADDI   r3 = r0 + 4
  LOAD   r6 = M[r3]

LOOP:
  MUL    r4 = r3 * r1
  ADD    r5 = r5 + r4

  ADDI   r1 = r1 + 1
  BRANCH r1 <= 10, LOOP
```

Loop Optimization

- **Induction variable analysis and elimination** - i is an induction variable if only definitions of i within loop increment/decrement i by loop-invariant value.
- **Strength reduction** - replace expensive instructions (like multiply) with cheaper ones (like add).

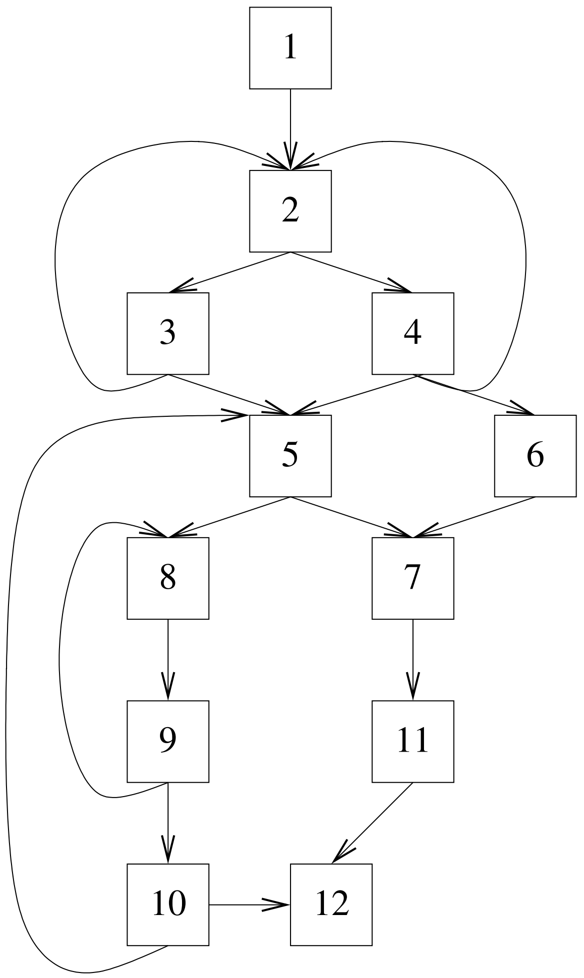
```
  ADDI   r1 = r0
  LOAD   r2 = M[FP + a]
  ADDI   r3 = r0 + 4
  LOAD   r6 = M[r3]
  SUB    r7 = r1 - 1
  MUL    r4 = r3 * r7
LOOP:
  ADD    r4 = r4 + r3
  ADD    r5 = r5 + r4
  ADDI   r1 = r1 + 1
  BRANCH r1 <= 10, LOOP
```

Back Edges



- *Back-edge* - flow graph edge from node n to node h such that h dominates n
- Each back-edge has a corresponding *natural loop*.

Natural Loops



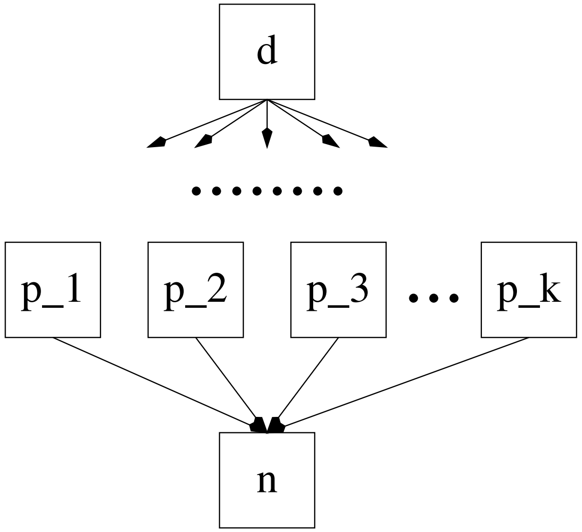
- Natural loop of back-edge $\langle n, h \rangle$:
  - has a loop header h.
  - set of nodes X such that h dominates $x \in X$ and there is a path from x to n not containing h.
- A node h may be header of more than one natural loop.
- Natural loops may be nested.
- Some loop optimizations (loop invariant code removal) need to insert statements immediately before loop header.
  - Create a loop *preheader* - a basic block before the loop header block.

Basic Blocks

- A *basic block* is a sequence of instructions that has one entry and one exit point.
- In assembly language:
  - Basic blocks begin with a `LABEL`.
  - Last statement of basic block is `JUMP, CJUMP`.
  - No `LABEL, JUMP, CJUMP` statements in between.
- Basic blocks are effective units of analysis or optimization. All instructions in a basic block can be grouped together.

Domination

- Assume every Control Flow Graph (CFG) has *start* node s_0 with no predecessors.
- Node d *dominates* node n if every path of directed edges from s_0 to n must go through d.
- Every node dominates itself.
- Consider:



- If d dominates each of the p_i, then d dominates n.
- If d dominates n, then d dominates each of the p_i.

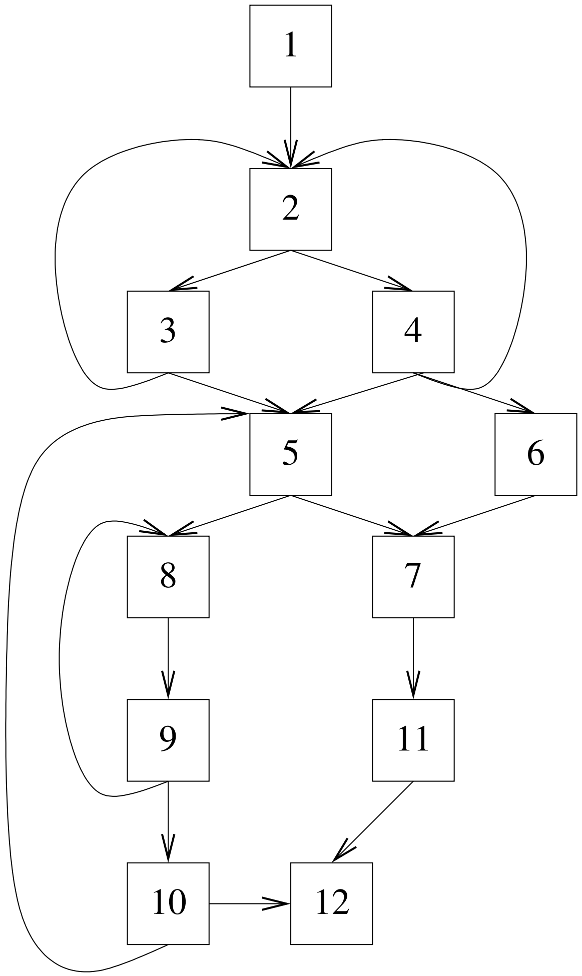
Dominator Analysis

- $Dom[n]$ = set of nodes that dominate node n.
- N = set of all nodes.
- For each node n in N, must compute $Dom[n]$

Dominator Analysis

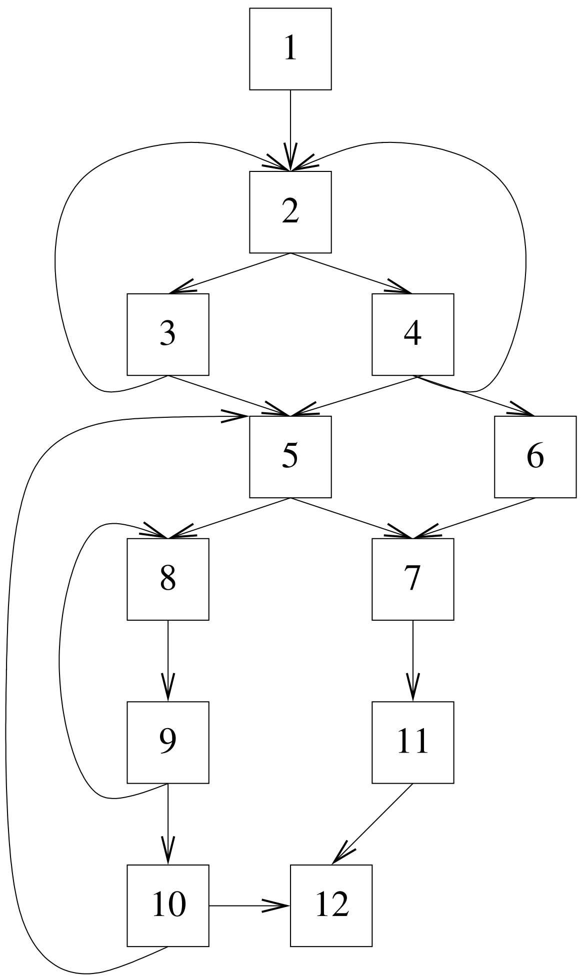
- If d dominates each of the predecessors p_i, then d dominates n.
- $Dom[n]$ = set of nodes that dominate node n.
- N = set of all nodes.
- Computation:
  1. $Dom[s_0] = \{s_0\}$.
  2. **for** $n \in N - \{s_0\}$ **do** $Dom[n] = N$
  3. **while** (changes to any $Dom[n]$ occur) **do**
  4. **for** $n \in N - \{s_0\}$ **do**
  5. $Dom[n] = \{n\} \cup \left(\bigcap_{p \in pred[n]} Dom[p]\right)$.

Dominator Analysis Example



Node	$Dom[n]$	$Dom[n]$
1	1	
2	1-12	
3	1-12	
4	1-12	
5	1-12	
6	1-12	
7	1-12	
8	1-12	
9	1-12	
10	1-12	
11	1-12	
12	1-12	

Dominator Example

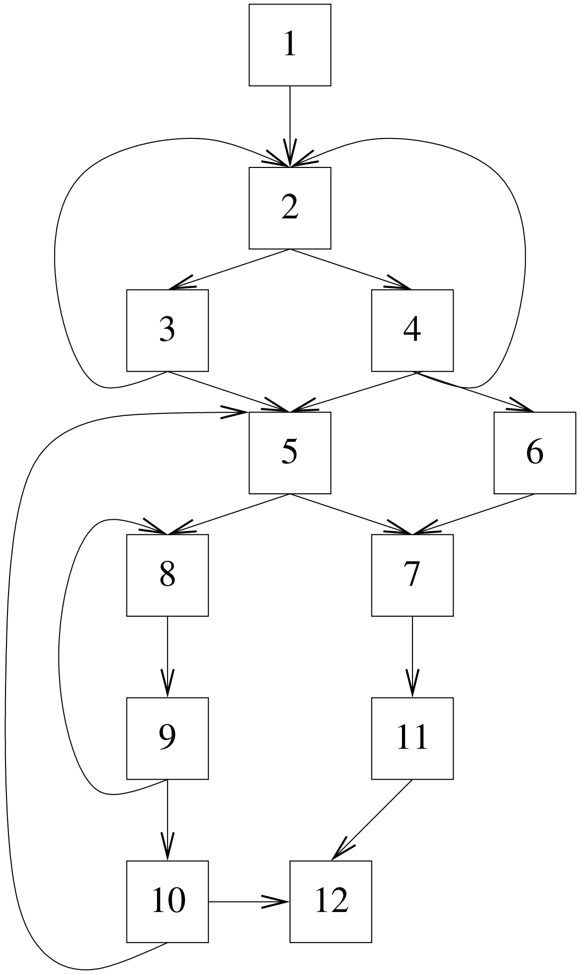


Node	$Dom[n]$
1	1
2	1,2
3	1,2,3
4	1,2,4
5	1,2,5
6	1,2,4,6
7	1,2,7
8	1,2,5,8
9	1,2,5,8,9
10	1,2,5,8,9,10
11	1,2,7,11
12	1,2,12

Immediate Dominator

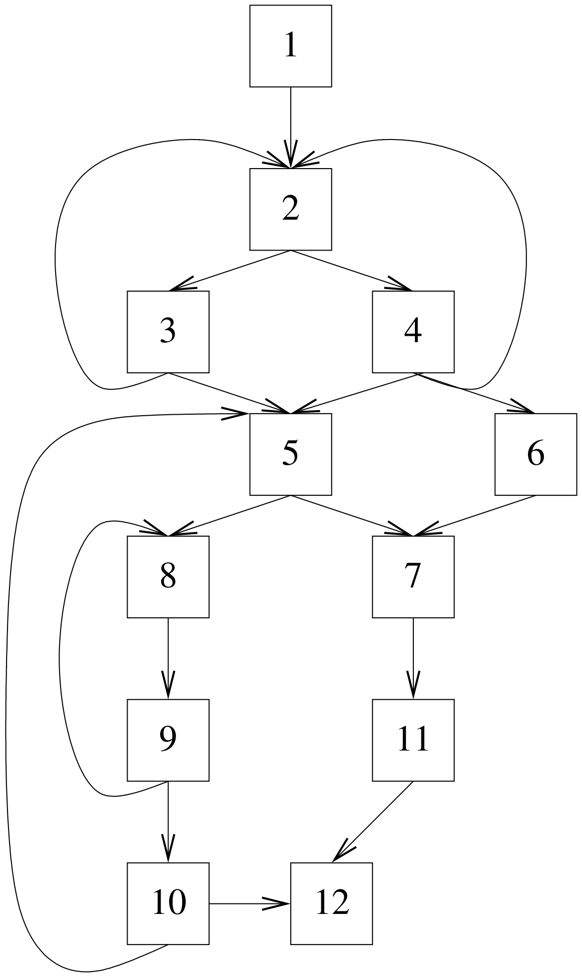
- Immediate dominator used in constructing *dominator tree*.
- Dominator Tree:
  - efficient representation of dominator information
  - helps us find loops
- s_0 is root of dominator tree.
- Each node d dominates only its descendants in tree.
- Every node n ($n \neq s_0$) has exactly one immediate dominator $IDom[n]$.
- $IDom[n] \neq n$
- $IDom[n]$ dominates n
- $IDom[n]$ does not dominate any other dominator of n.
- Last dominator of n on any path from s_0 to n is $IDom[n]$.

Immediate Dominator Example



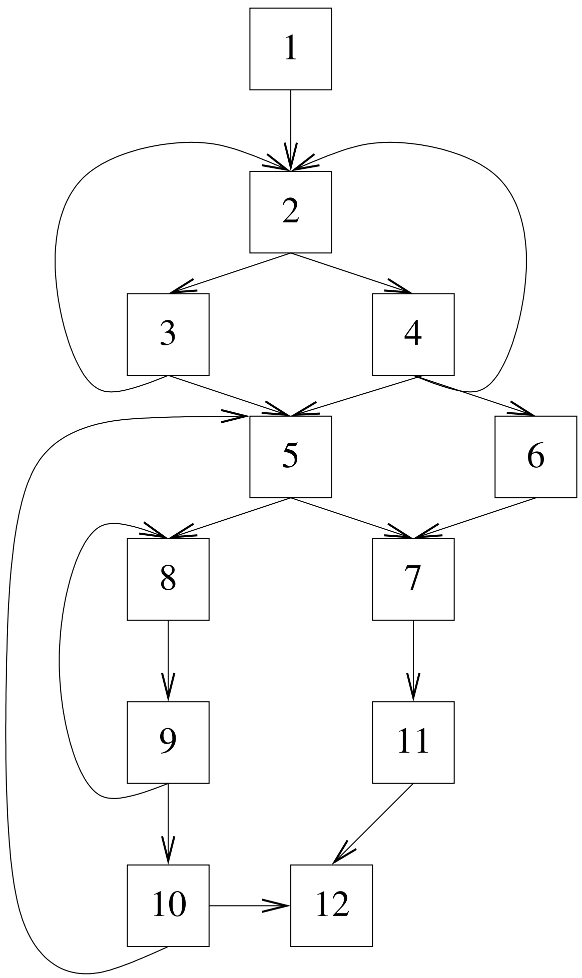
Node	$Dom[n]$	$IDom[n]$
1	1	
2	1,2	
3	1,2,3	
4	1,2,4	
5	1,2,5	
6	1,2,4,6	
7	1,2,7	
8	1,2,5,8	
9	1,2,5,8,9	
10	1,2,5,8,9,10	
11	1,2,7,11	
12	1,2,12	

Immediate Dominator Example



Node	$Dom[n]$	$IDom[n]$
1	1	
2	1,2	1
3	1,2,3	2
4	1,2,4	2
5	1,2,5	2
6	1,2,4,6	4
7	1,2,7	2
8	1,2,5,8	5
9	1,2,5,8,9	8
10	1,2,5,8,9,10	9
11	1,2,7,11	7
12	1,2,12	2

Finding Loops: Summary



Nested loops: if A and B are loops with headers a and b such that a is not b and a dominates b then the nodes of B are a proper subset of the nodes of A. We say that B is a loop nested within A.

Computing the loop nest of a graph G:

- compute dominators of G
- construct the dominator tree.
- consider each edge e in the graph
  - is e a back edge? (Does it point to one of its dominators?)
  - if so, it forms a natural loop; remember the header
- for each loop header h, merge all natural loops of h into a single loop. $loop[h]$
- construct the tree of loop headers such that h_1 is above h_2 if h_1 dominates h_2

Loop Optimizations

once we have identified the loops and inner loops, we can attempt to focus our optimization efforts on them.

- Avoiding spilling registers inside loops
- Hoisting loop invariant computations
- Strength reduction
- Loop unrolling

Loop Invariant Computations

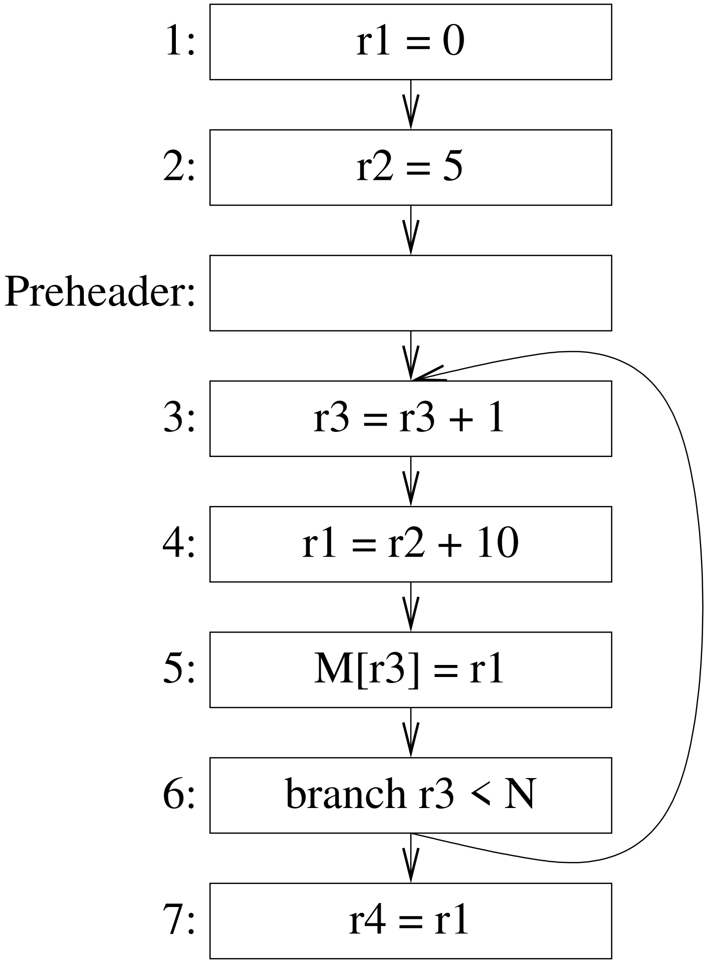
- Given statements in loop s: `t = ` a_1 `op` a_2:
  - s is loop-invariant if a_1, a_2 have same value each loop iteration.
  - may sometimes be possible to hoist s outside loop.
- Cannot always tell whether a will have same value each iteration $\rightarrow$ conservative approximation.
- d: `t = ` a_1 `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 Invariant Computation: Algorithm

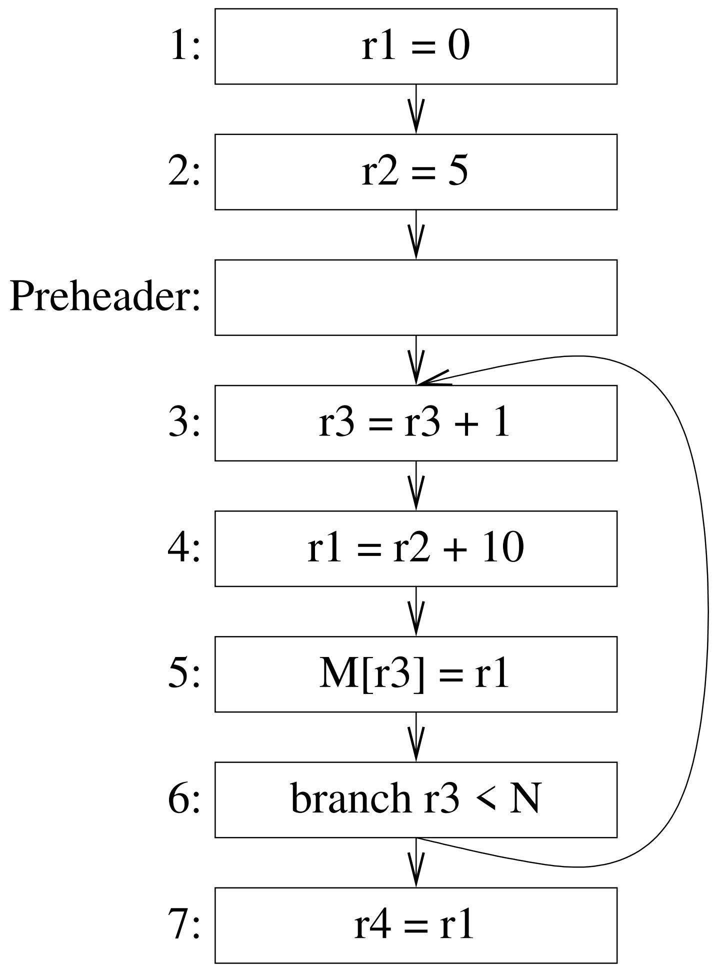
Iterative algorithm for determining loop-invariant computations:

- 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
  - have a single reaching definition in loop marked invariant.

Loop Invariant Code Dectection



Loop Invariant Code Dectection

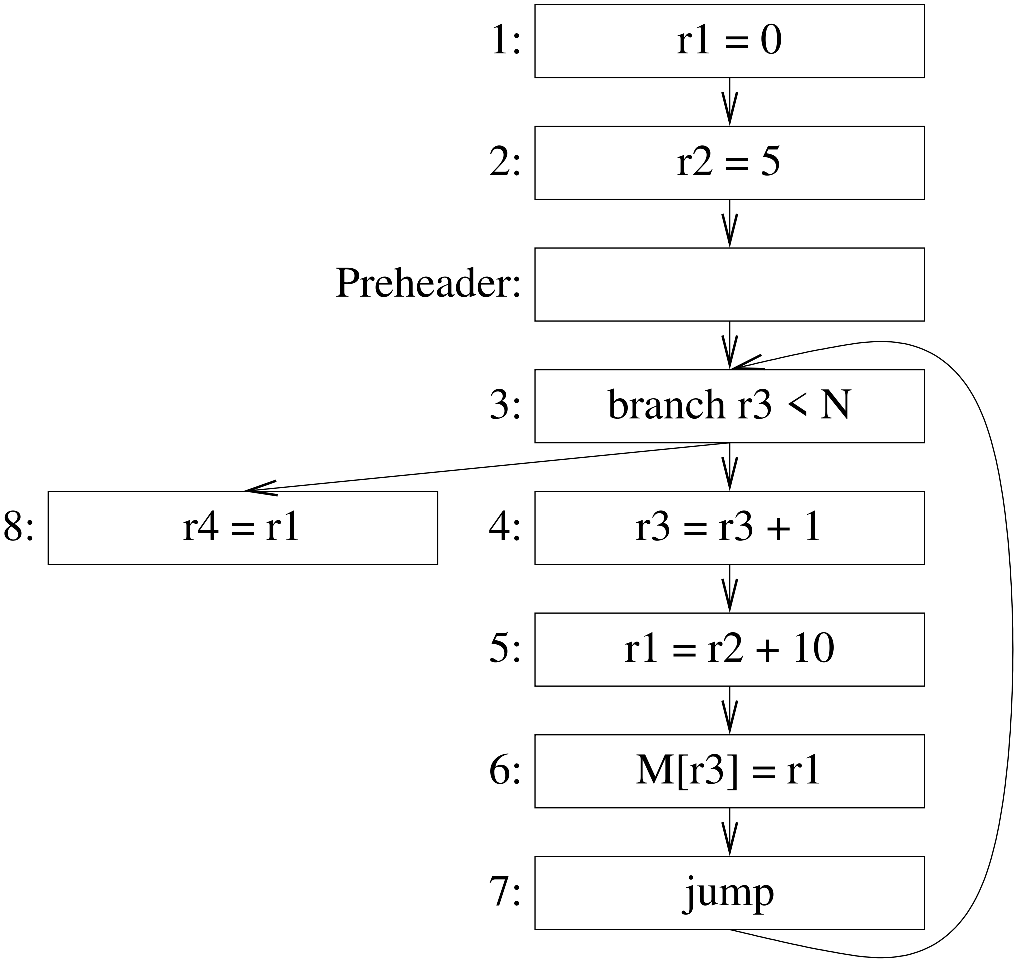


Hoist loop-invariant code to pre-header — **when possible**.

Loop Invariant Code Motion: Constraint 1

d: `t = a op b`

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



Loop Invariant Code Motion: Constraint 1

d: `t = a op b`

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

Problem with while loops: Loop exit dominates all nodes in the loop. eg:

```
while x > 0 {x = x - 1; ...}
```

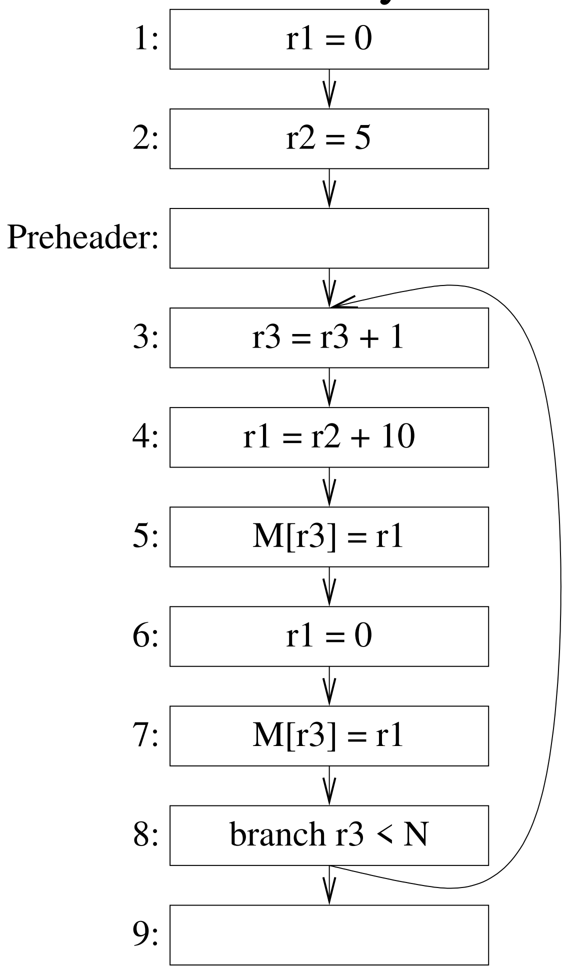
Solution: Transform into repeat:

```
if (x > 0) {
  x = x - 1; ...;
  repeat {x = x - 1; ...} until (x <= 0)
}
```

Loop Invariant Code Motion: Constraint 2

d: `t = a op b`

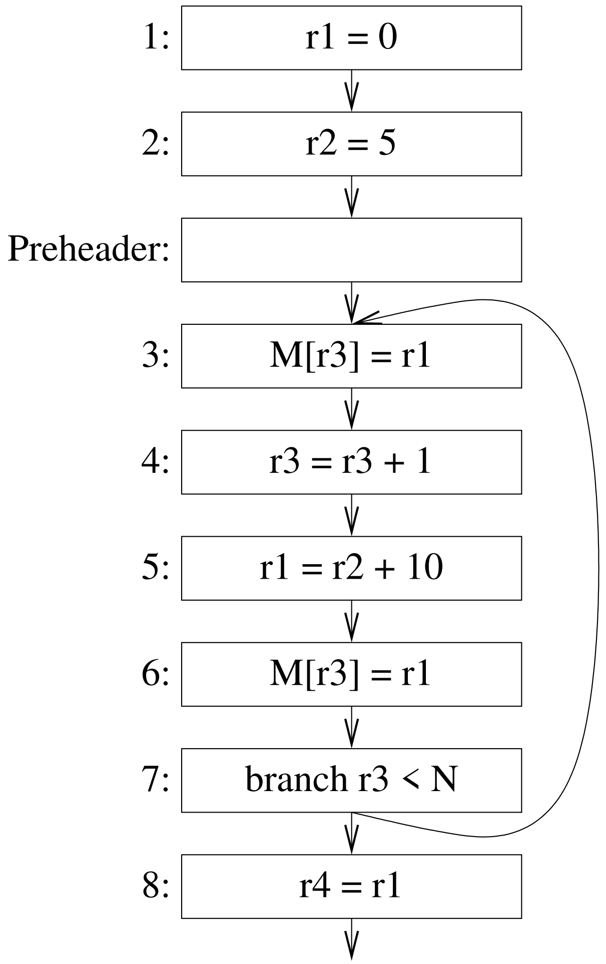
there must be only one definition of `t` inside loop.



Loop Invariant Code Motion: Constraint 3

d: `t = a op b`

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



Loop Invariant Code Motion

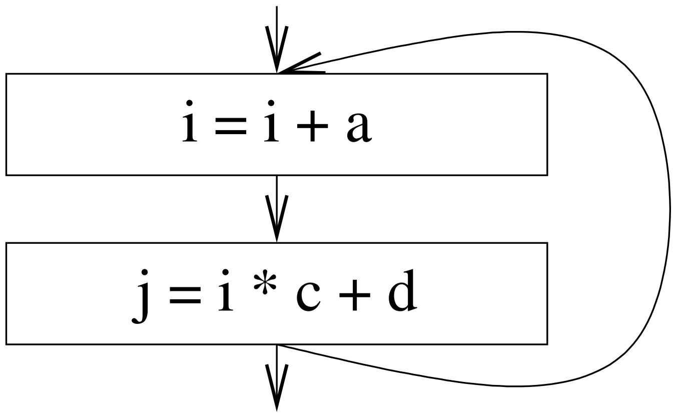
Algorithm for code motion:

- 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 an **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**

Induction Variables

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.
- 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`

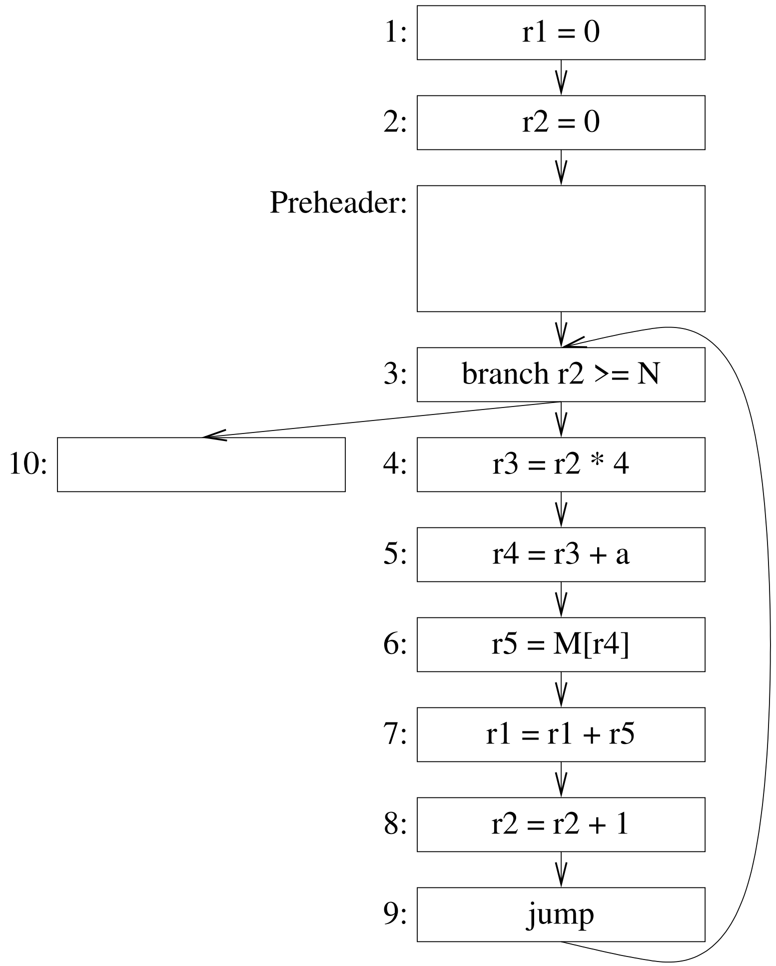
Induction Variable Detection: Algorithm

Algorithm for induction variable detection:

- 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` $\longrightarrow$ `(i, a*c, b*c)`
  - `k = j + d` $\longrightarrow$ `(i, a + d, b)`

Induction Variable Detection: Example

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


1:
r1 = 0
2:
r2 = 0
Preheader:
3:
branch r2 >= N
10:
4:
r3 = r2 * 4
5:
r4 = r3 + a
6:
r5 = M[r4]
7:
r1 = r1 + r5
8:
r2 = r2 + 1
9:
jump

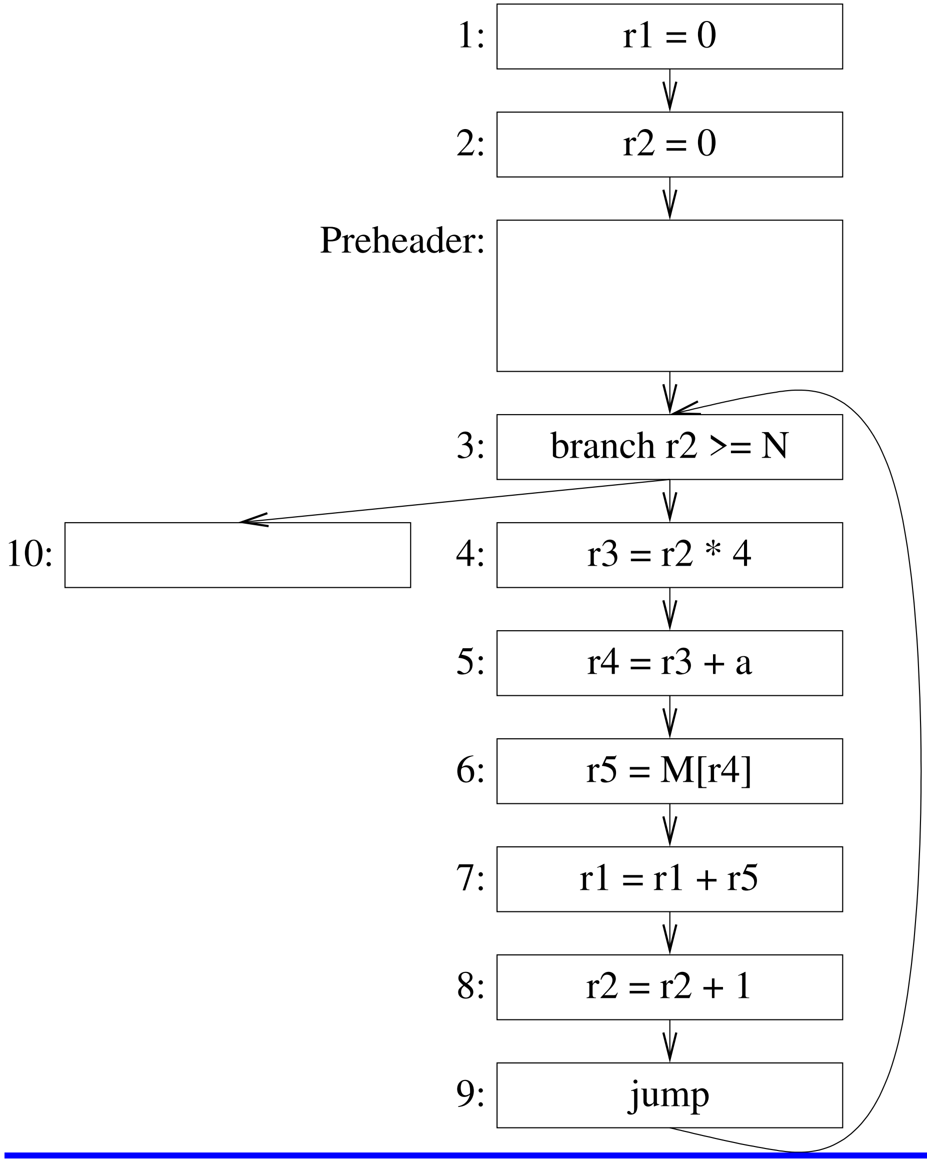
Strength Reduction

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:

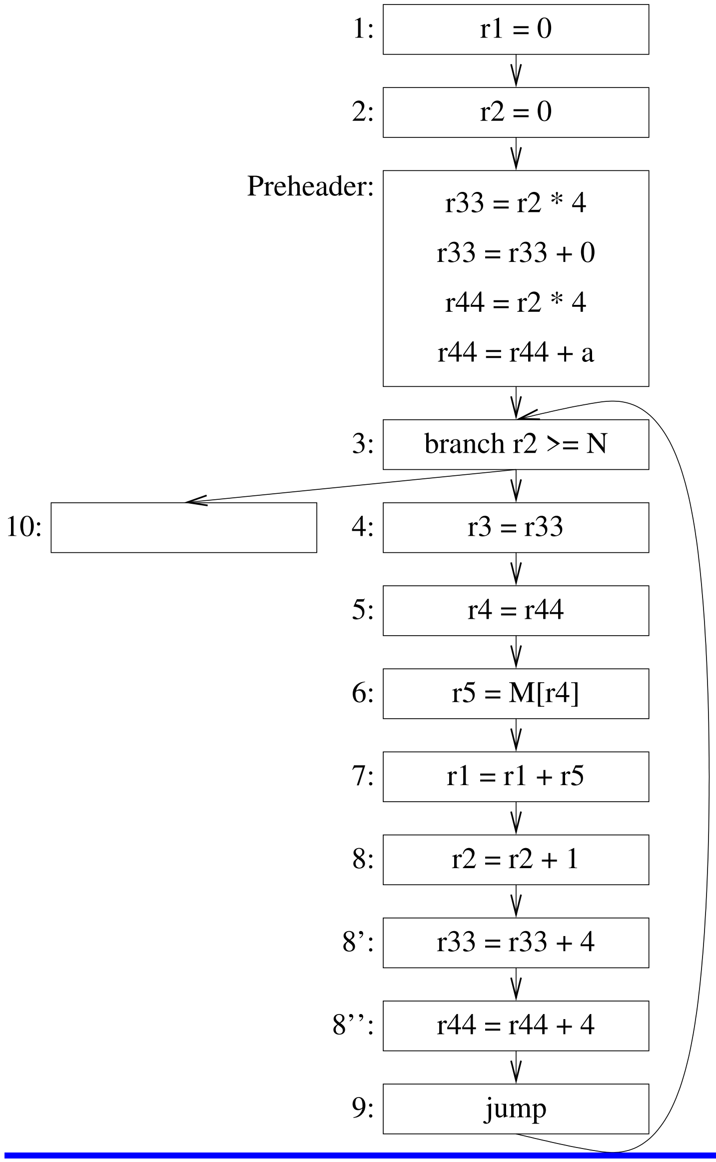
```
j' = b * i
j' = j' + a
```

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

Strength Reduction: Example



Strength Reduction: Example

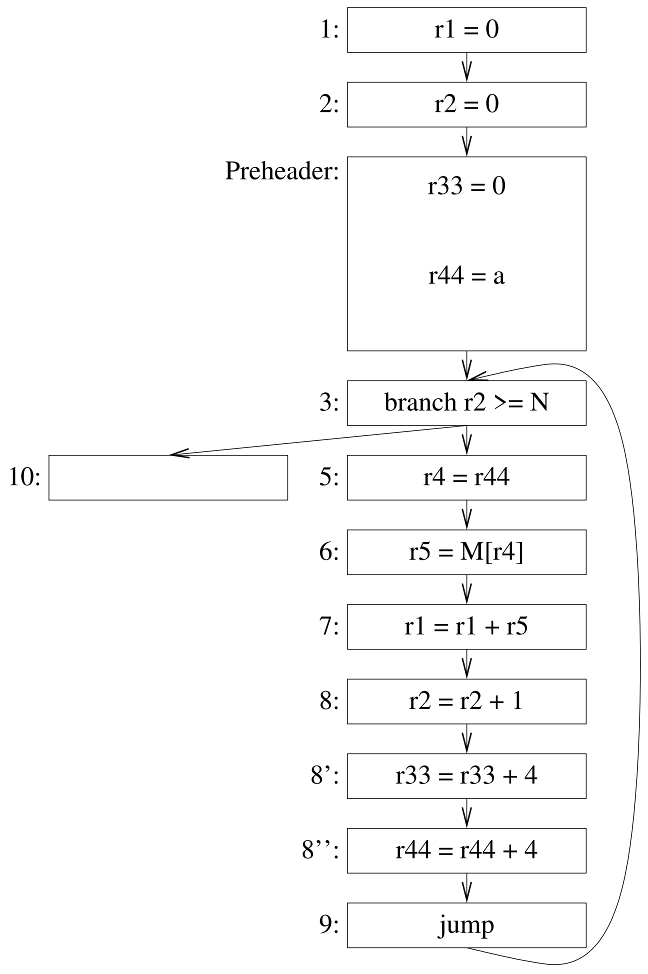


Induction Variable Elimination

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.

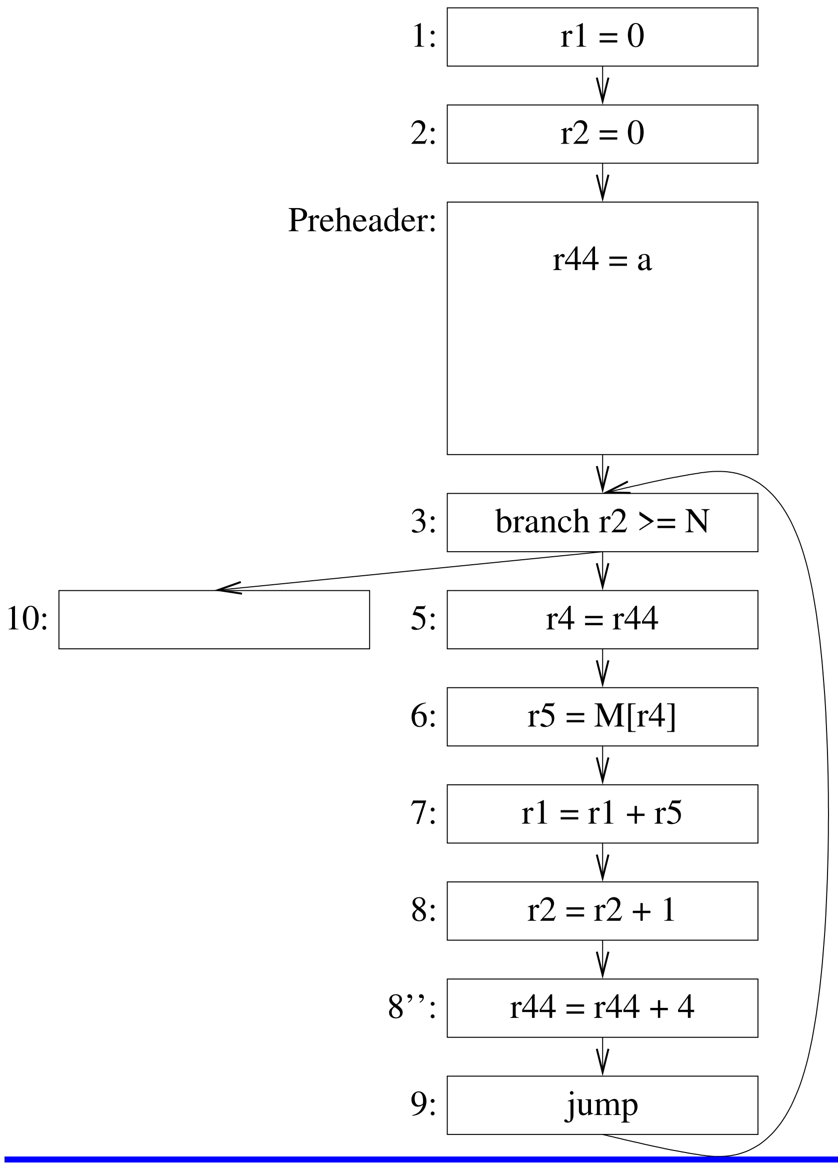
Induction Variable Elimination: Example



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