Topic 11: Loops

COS 320

Compiling Techniques

Princeton University
Spring 2015

Prof. David August

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: $\text{t} = a_1 \text{ op } a_2$:

    - 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: $\text{t} = a_1 \text{ op } a_2$ 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:
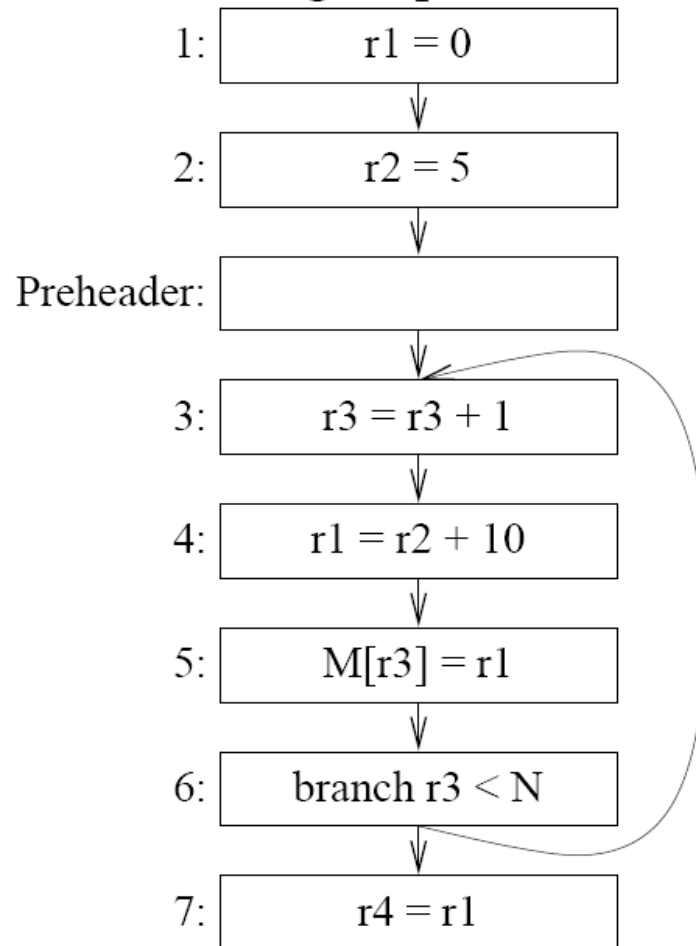
```
mark "invariant" all definitions whose operands
- are constant, or
- whose reaching definitions are outside loop.

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.
```

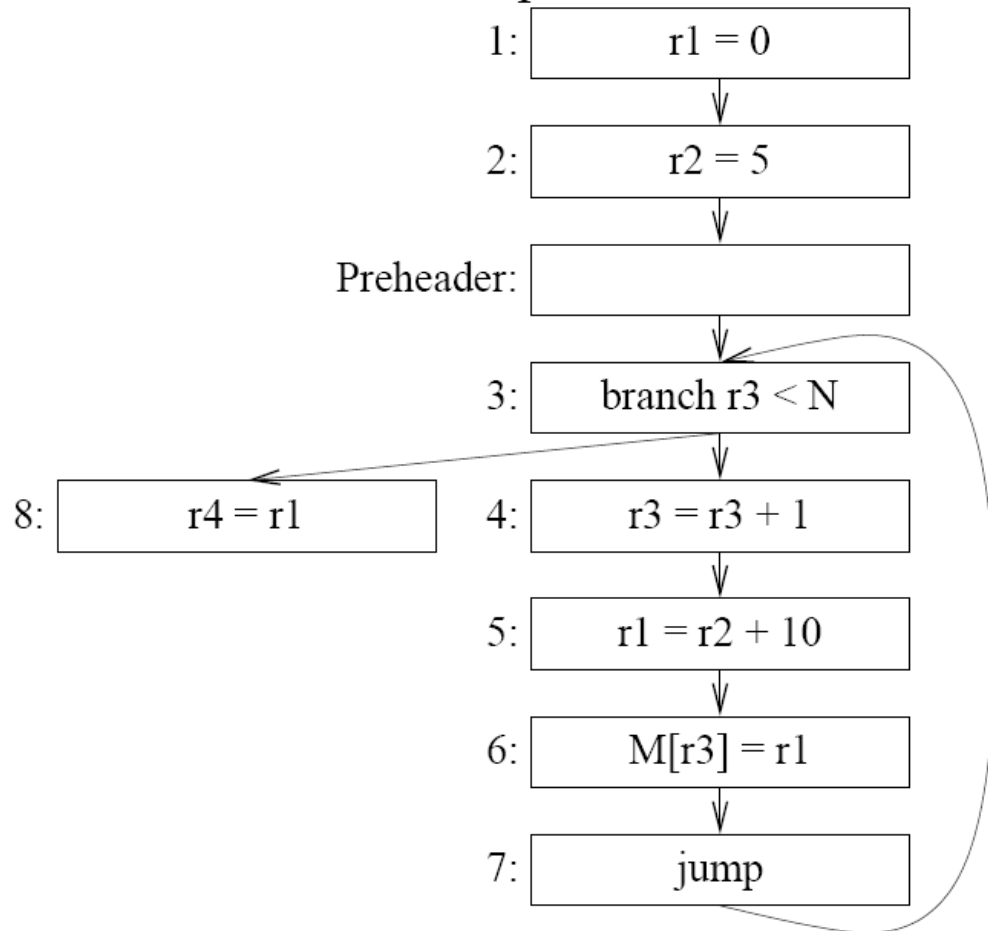After detecting loop-invariant computations, perform code motion.

1: | r1 = 0

2: | r2 = 5

Preheader: |

3: | r3 = r3 + 1

4: | r1 = r2 + 10

5: | M[r3] = r1

6: | branch r3 < N

7: | r4 = r1

Subject to some constraints.

d: t = a op b

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

```
1:  r1 = 0

2:  r2 = 5

Preheader:

3:  branch r3 < N

8:  r4 = r1      4:  r3 = r3 + 1

5:  r1 = r2 + 10

6:  M[r3] = r1

7:  jump
```

d: t = a op b

there must be only one definition of t inside loop.

1:	r1 = 0
2:	r2 = 5
Preheader:	
3:	r3 = r3 + 1
4:	r1 = r2 + 10
5:	M[r3] = r1
6:	r1 = 0
7:	M[r3] = r1
8:	branch r3 < N
9:	
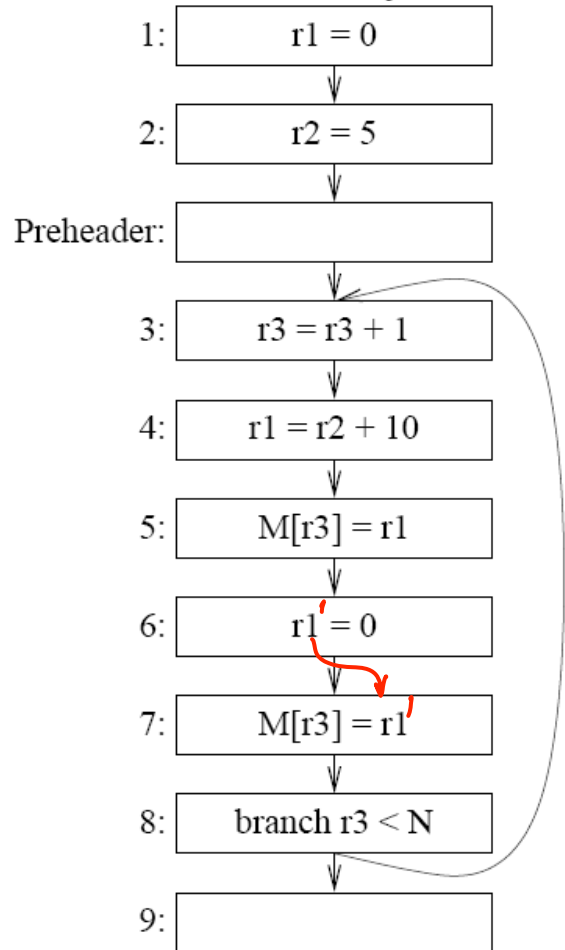
d: t = a op b

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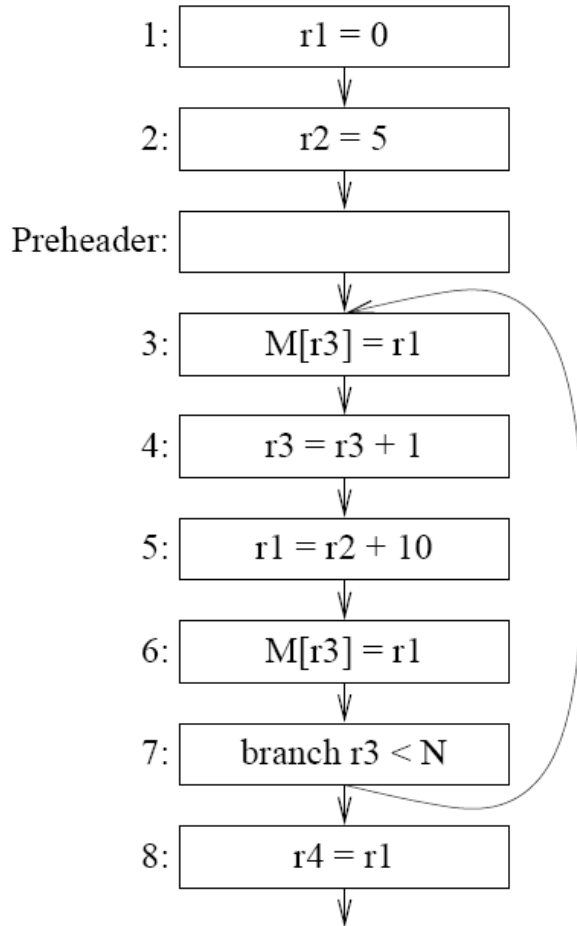
LICM

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.
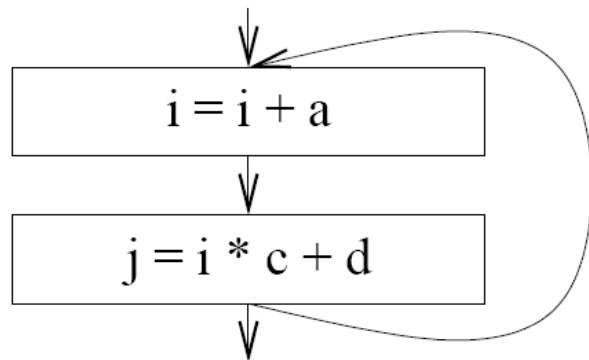
- `i` is an induction variable

- `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

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`   ← basic

- `i` is basic induction variable; `a` and `b` are loop invariant.

- value of `j` at point of definition is `a + b * i`
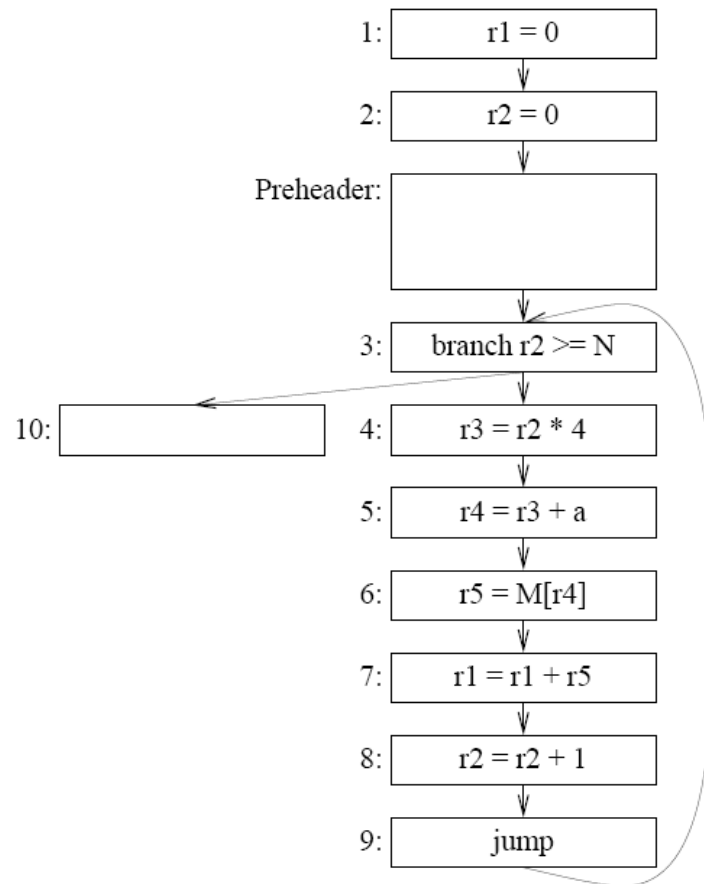
- `j` belongs to the family of `i`

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. 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`)

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


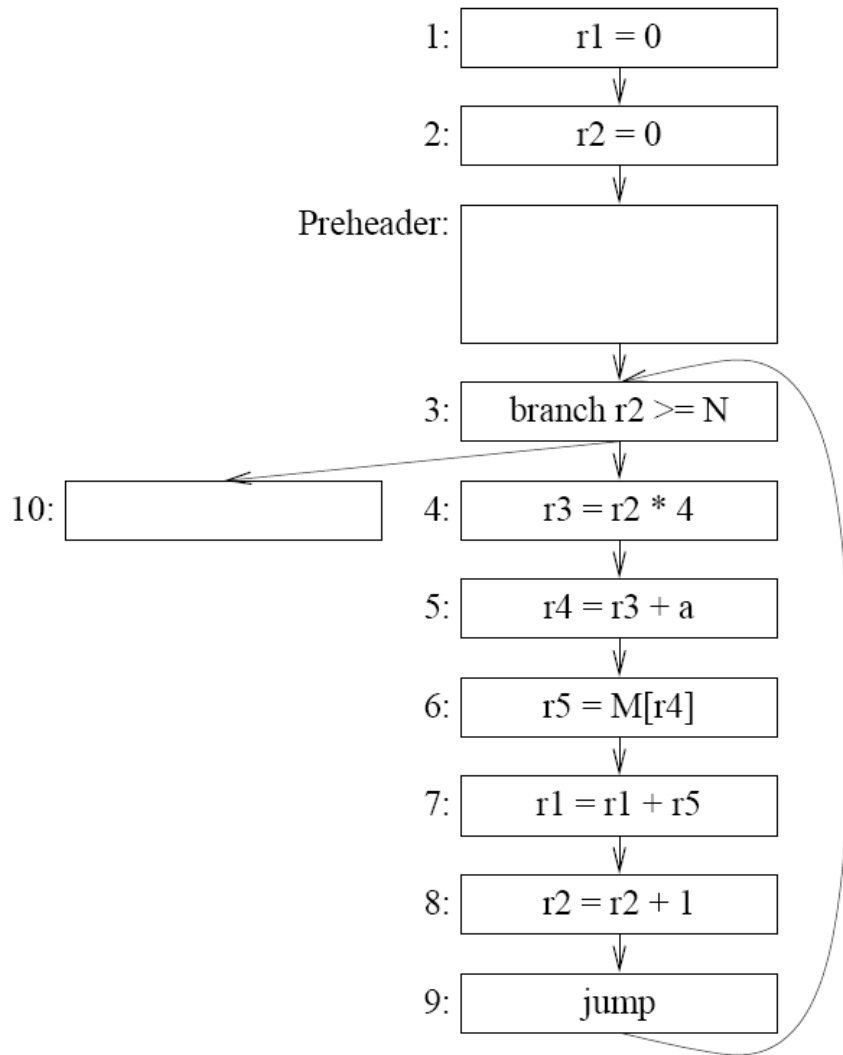|	
1:	r1 = 0
2:	r2 = 0
Preheader:	
3:	branch r2 >= N
4:	r3 = r2 * 4
5:	r4 = r3 + a
6:	r5 = M[r4]
7:	r1 = r1 + r5
8:	r2 = r2 + 1
9:	jump

10:

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'`.

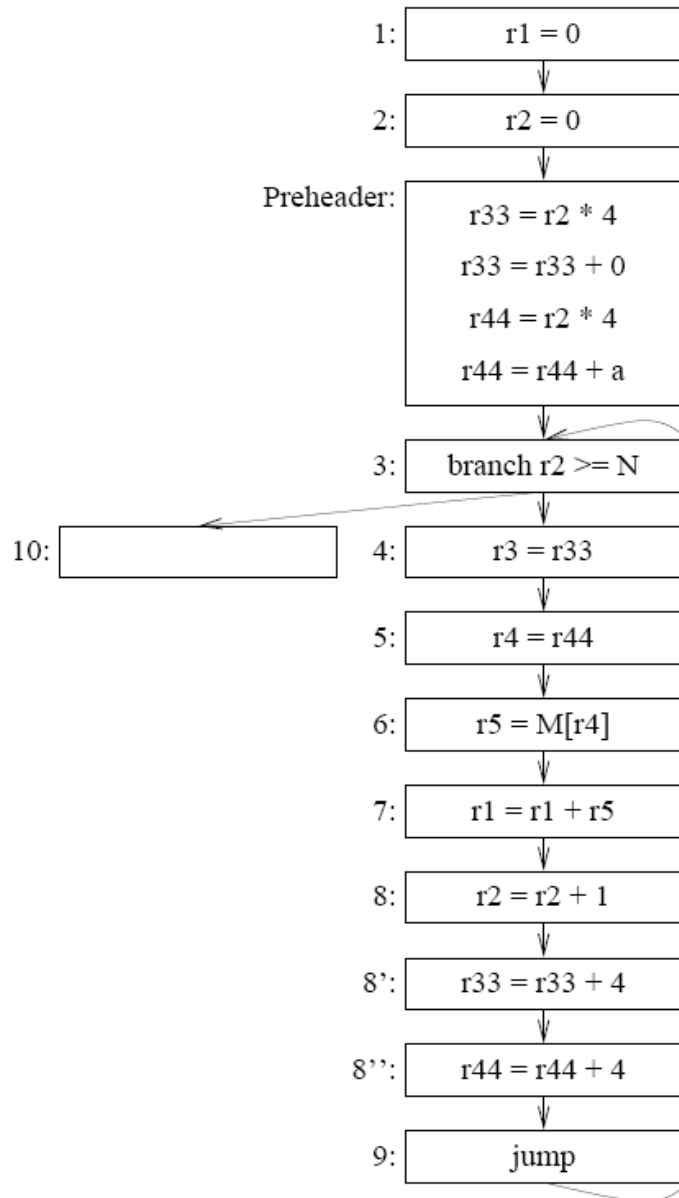4. Initialize `j'` at end of preheader node:

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

- Strength reduction still requires multiplication, but multiplication now performed outside loop.

- `j'` also has triple `(i, a, b)`
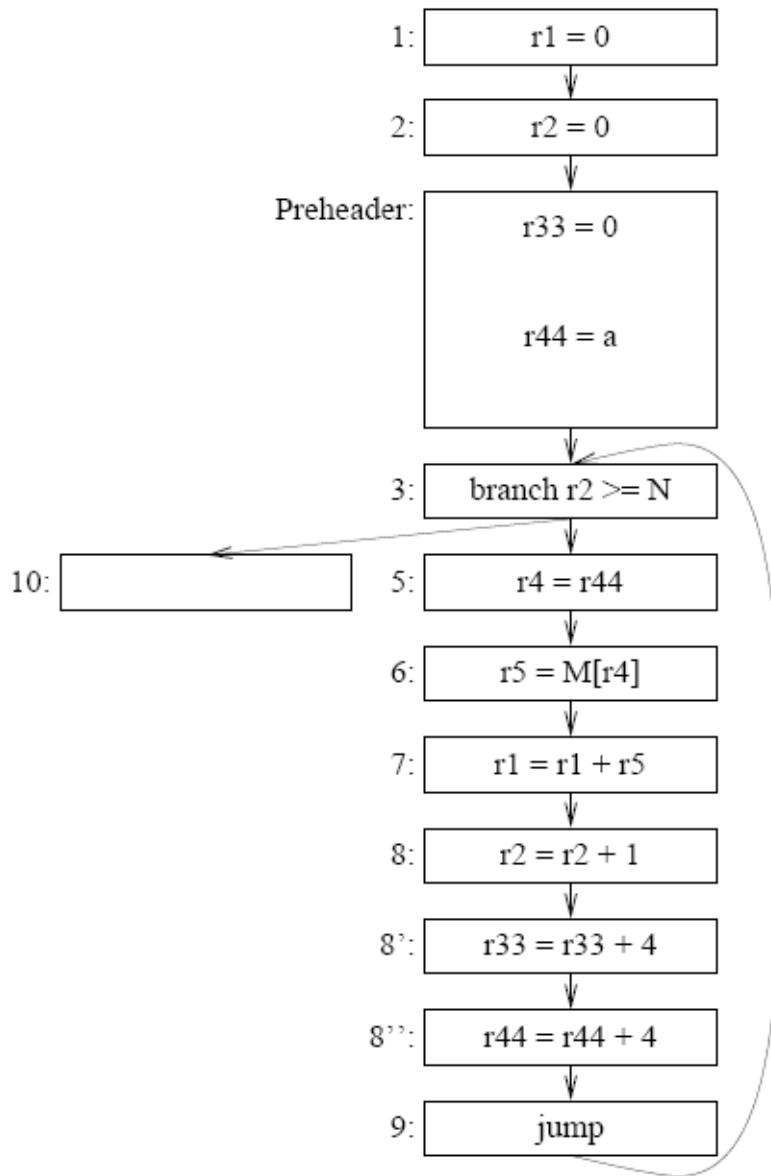
Strength Reduction Example

Strength Reduction Example

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.
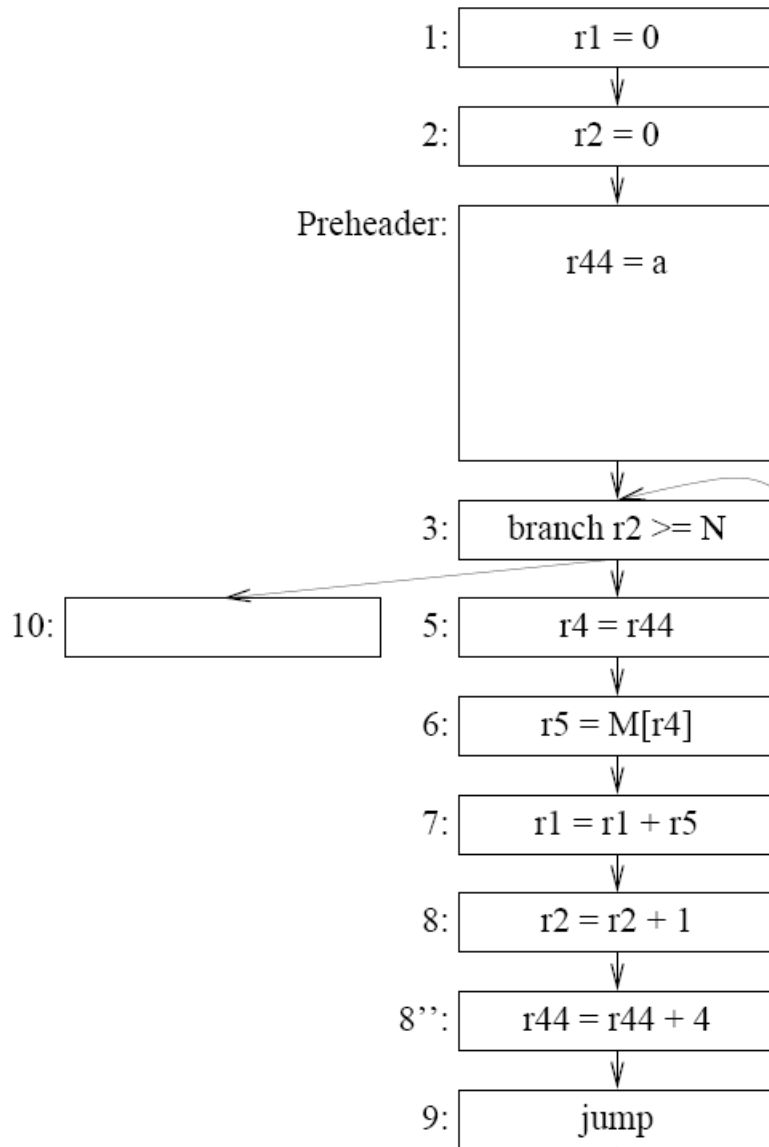
Induction Variable Elimination Example

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

1:    r1 = 0

2:    r2 = 0

Preheader: r44 = a

3:    branch r2 >= N

10:

5:    r4 = r44

6:    r5 = M[r4]

7:    r1 = r1 + r5

8:    r2 = r2 + 1

8'':   r44 = r44 + 4

9:    jump

Induction Variable Elimination: Example

1:	r1 = 0

2:	r2 = 0

Preheader:

r44 = a

r100 = 4 * N

r101 = r100 + a

3:	branch r44 >= r101

10:	

5:	r4 = r44

6:	r5 = M[r4]

7:	r1 = r1 + r5

8:	r2 = r2 + 1

8'':	r44 = r44 + 4

9:	jump