Where are we going?

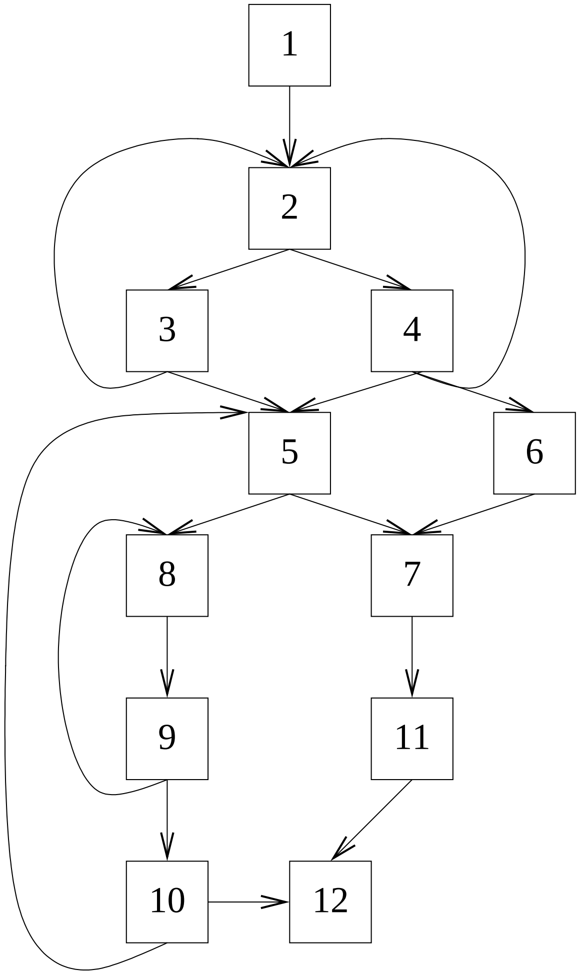
- Last time:
  - Started out on Chapter 18
  - What is a basic block? a dominator? a dominator tree? a loop? a loop header? a natural loop?
  - What kinds of loop optimizations can we do? Invariant code hoisting.
    - Find the invariant statements.
    - Check to see whether the invariant statements can be hoisted.
- Today:
  - Continue Chapter 18.
  - Review some definitions.
  - More loop optimizations.
    - Induction variable analysis, strength reduction and induction variable elimination.

Loops

- 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* 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* to h.
- A loop is a single entry, multiple exit region.

* Note: here, a path must have non-zero length and contain only nodes from S

Natural Loops



- *Back-edge* - flow graph edge from node n to node h such that h dominates n
- 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.

Finding Nested Loops

- Compute dominators.
- Compute dominator tree.
- Find the natural loops and therefore loop headers.
  - Traverse the dominator tree (from the most to least dominating)
  - For each dominator, find the back edges pointing to it.
  - For each back edge, find the rest of nodes in the loop.
- Merge loops that share the same loop header. Call *loop[h]* the set of nodes in the loop for header h.
- Compute the loop nest tree.
  - Traverse the dominator tree and create a new tree with one node for each loop header.
  - If $h' \in loop[h]$ then draw an arc from h to h'.

Once we have the loop nest tree, we start optimizing from the leaves.

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).

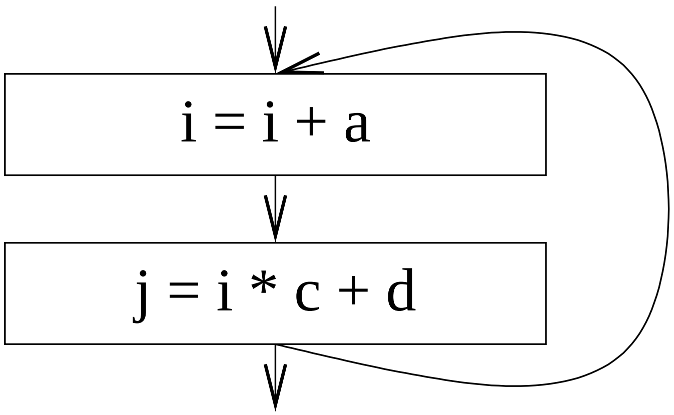
```
  ADDI   r1 = r0 + 0
  LOAD   r2 = M[FP + a]
  ADDI   r3 = r0 + 4
  LOAD   r6 = M[FP + x]

LOOP:
  MUL    r4 = r3 * r1
  ADD    r5 = r2 + r4
  STORE  M[r5] = r6

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

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` ⇒ strength reduction
  - may not need to use `i` in loop ⇒ induction variable elimination

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:

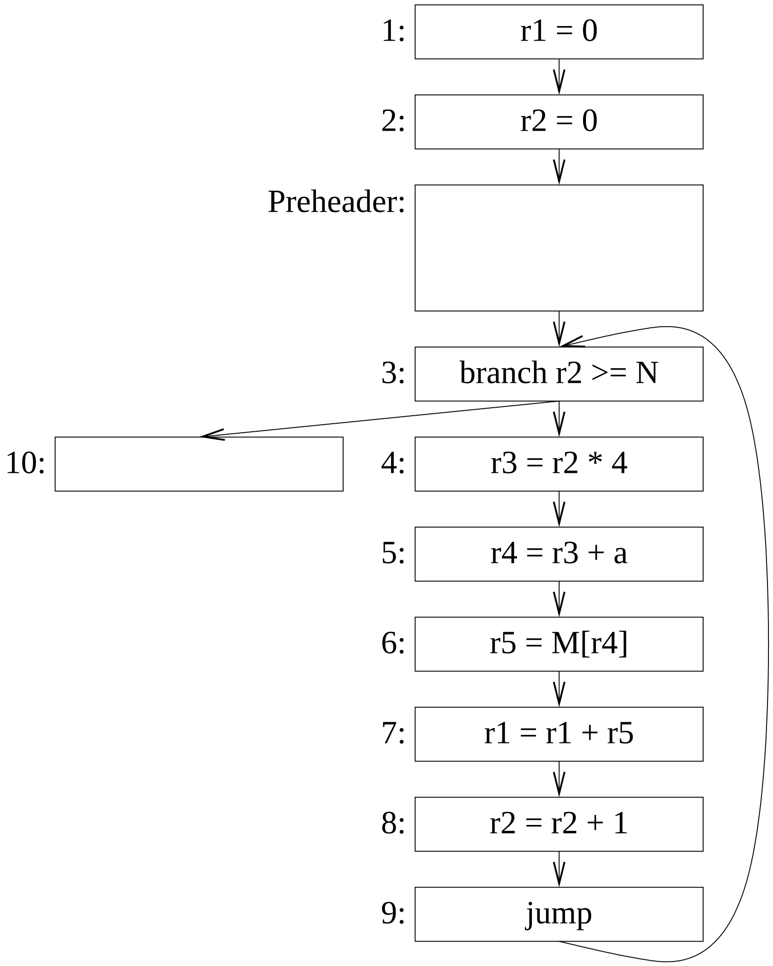
- Part 1: Scan statements of L for basic induction variables `i`
  - for each `i`, associate triple `(i, 0, 1)`
  - `i` belongs to its own family.

Induction Variable Detection: Algorithm

- Part 2: 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` $\rightarrow$ `(i, a*c, b*c)`
  - `k = j + d` $\rightarrow$ `(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

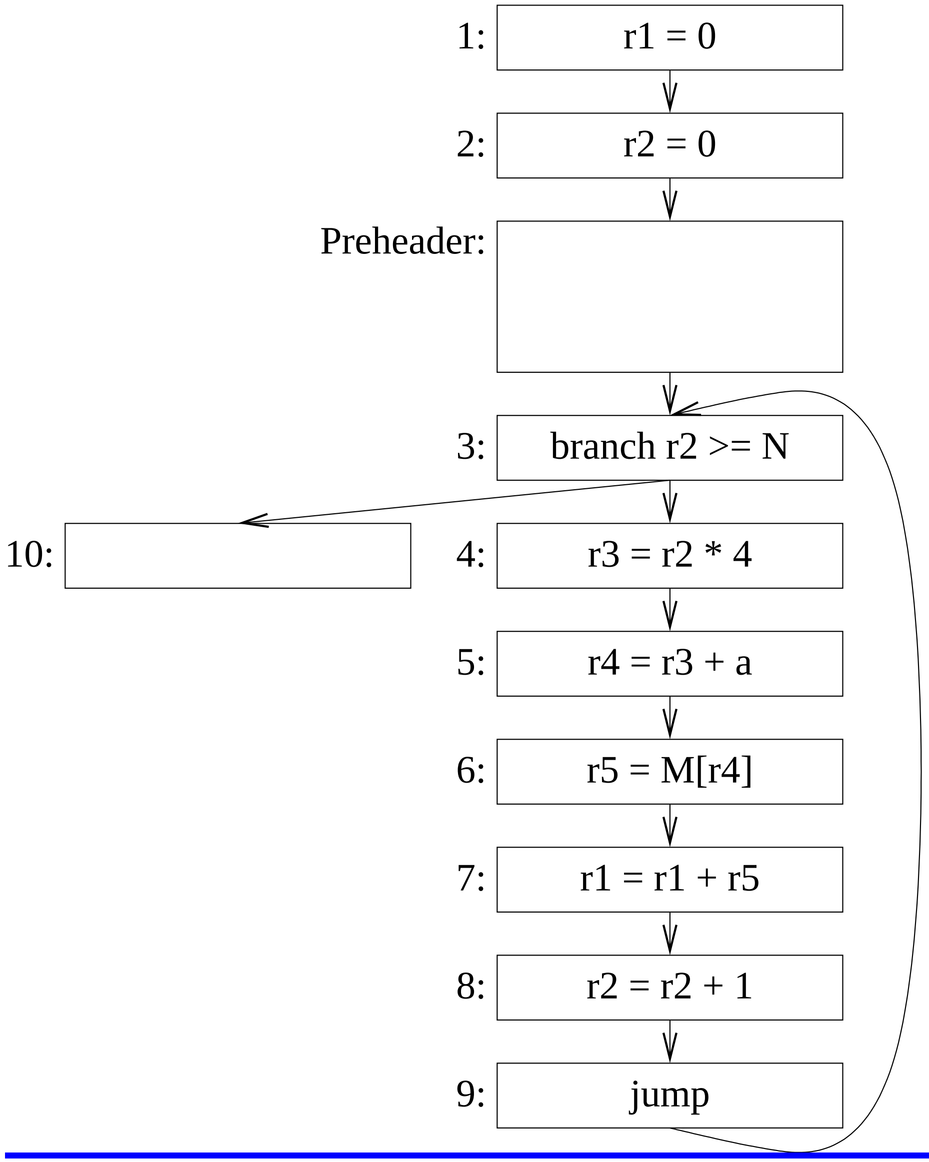
Strength Reduction

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:

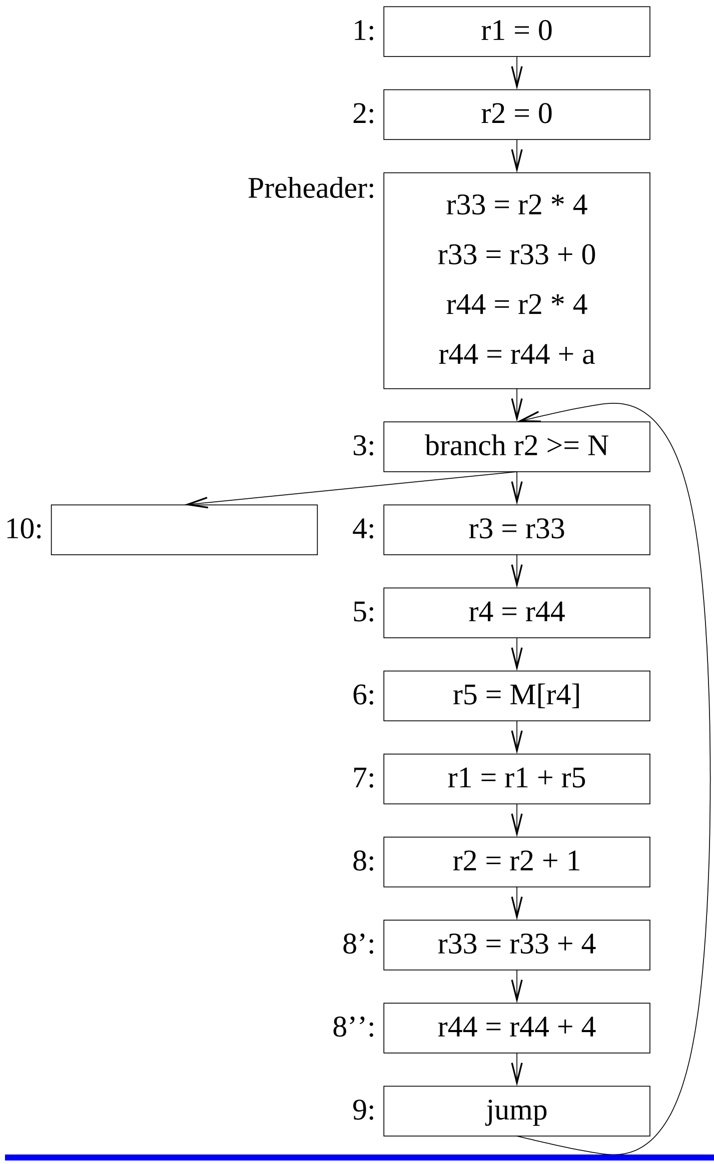
```
j' = b * i
j' = j' + a
```

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

Strength Reduction: Example



Strength Reduction: Example

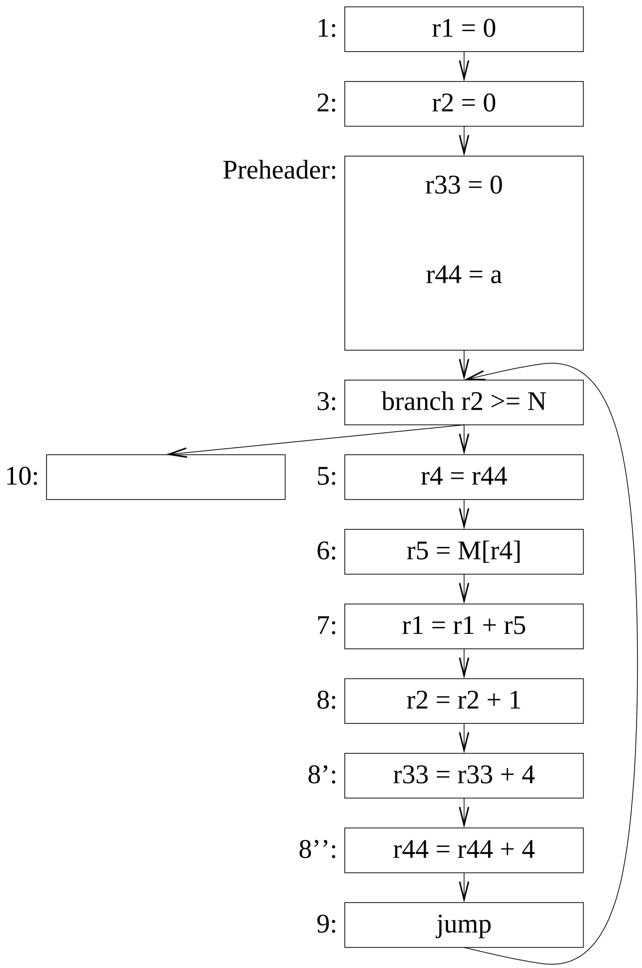


Induction Variable Elimination

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.

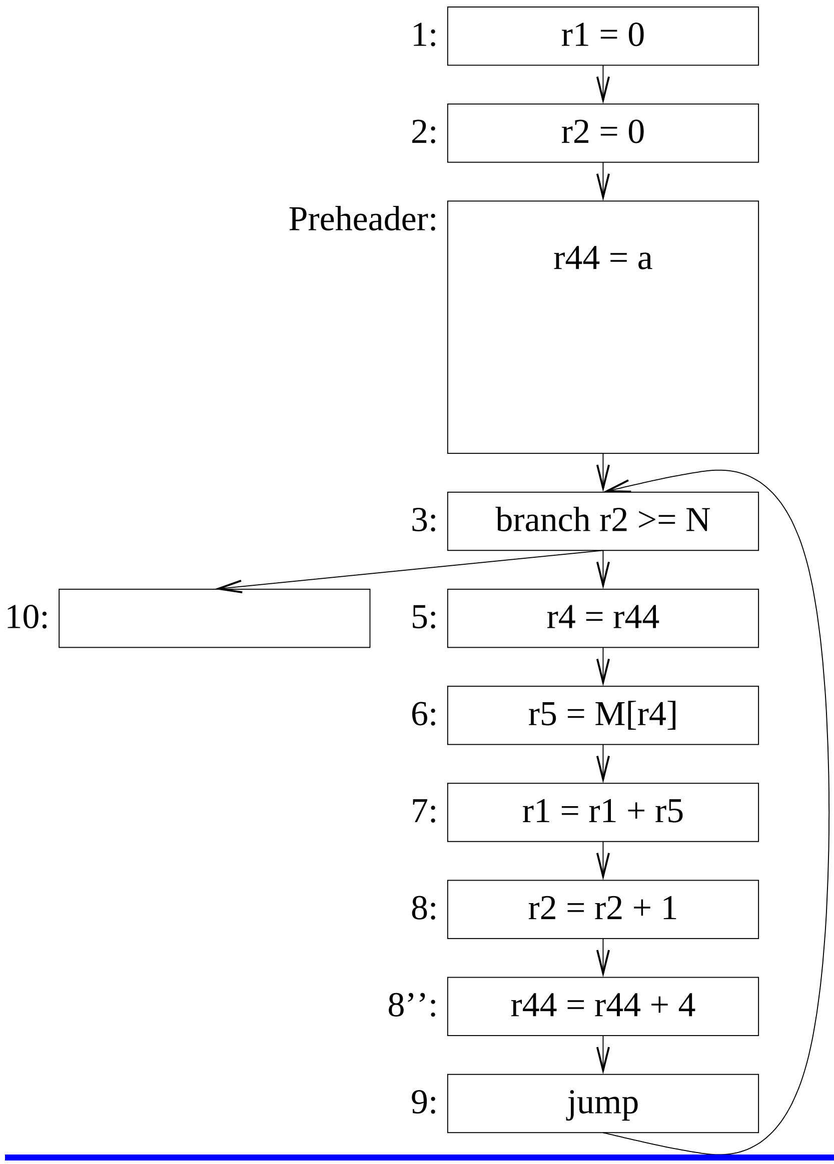
Induction Variable Elimination: Example



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