Lecture 8:

Induction Variable Optimizations

I. Finding loops

II. Overview of Induction Variable Optimizations

III. Further details

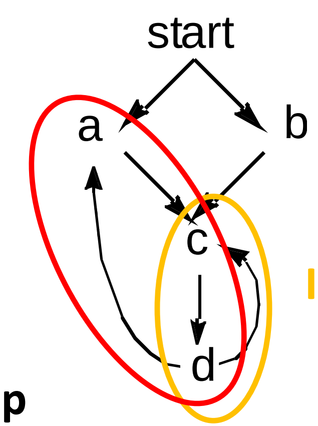
ALSU 9.1.8, 9.6, 9.8.1

What is a Loop?

- **Goals:**
  - Define a loop in graph-theoretic terms (control flow graph)
  - Independent of specific programming language constructs used
  - A uniform treatment for all loops: DO, while, for, goto’s
- **Not every cycle is a “loop” from an optimization perspective**

Is this a loop?

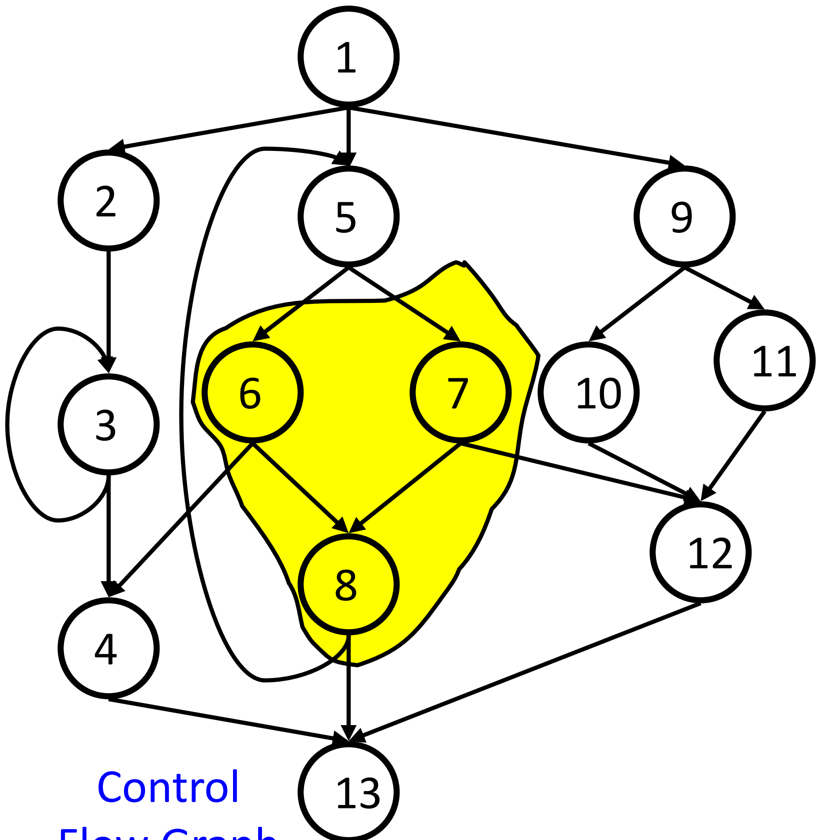
no



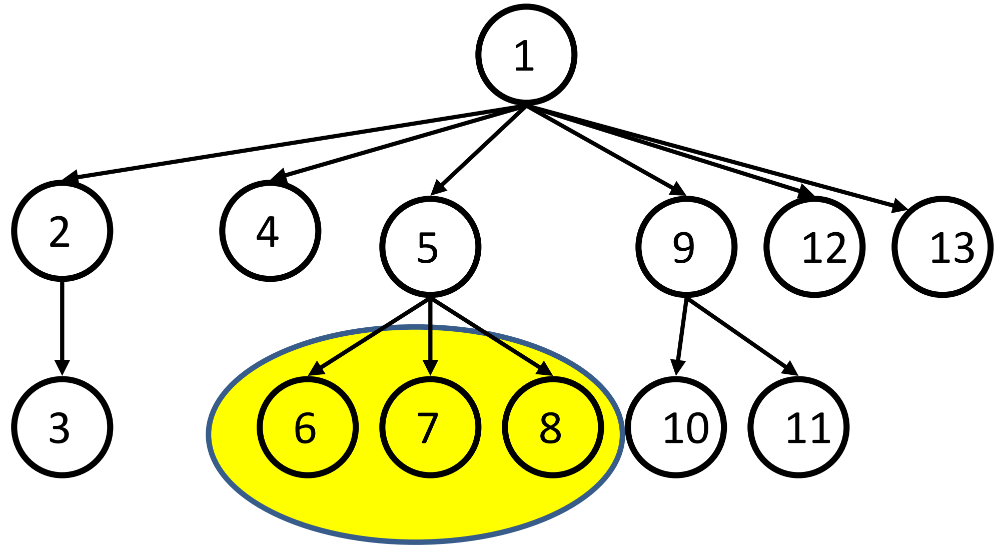
Is this a loop? **yes**

- **Intuitive properties of a loop**
  - single entry point
  - edges must form at least a cycle
- **Loops can nest**

Important Concept: Dominance



Control
Flow Graph



Dominance Tree (D-Tree)

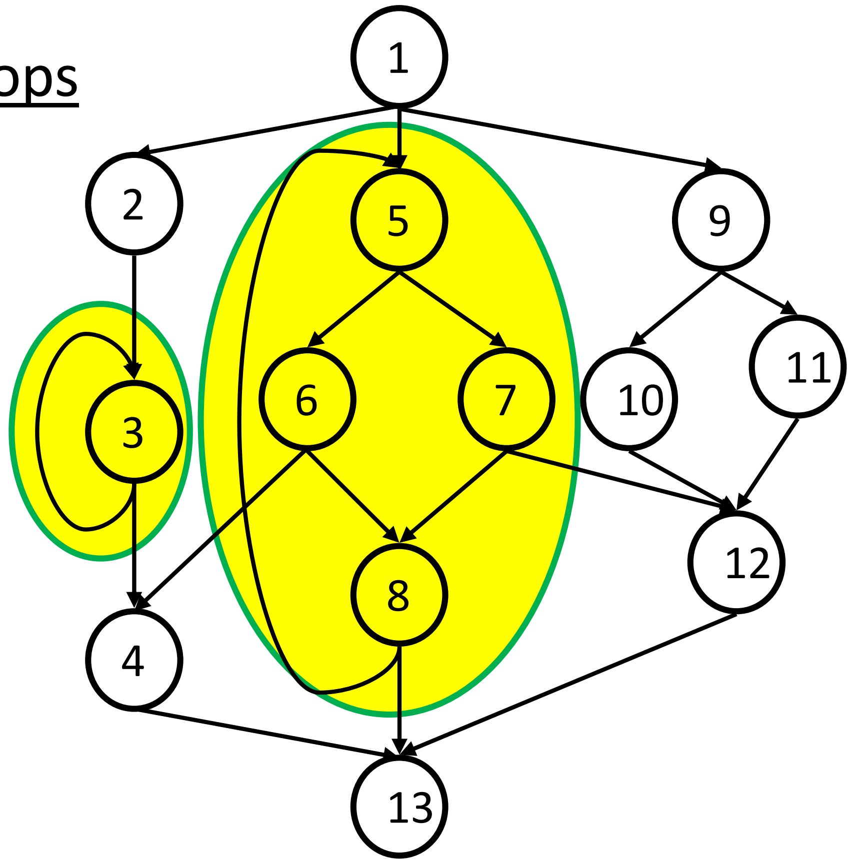
x sdom w iff x is a proper ancestor of w

All paths to 6, 7, or 8 must visit 5 first

x strictly dominates w (x sdom w) iff impossible to reach w without passing through x first

x dominates w (x dom w) iff x sdom w OR x = w

Natural Loops



- Single entry-point: ***header***
  - a header dominates all nodes in the loop
- A ***back edge*** is an arc t->h whose head h dominates its tail t
  - a back edge must be a part of at least one loop
- The **natural loop of a back edge** t->h is the smallest set of nodes that includes t and h, and has no predecessors outside the set, except for the predecessors of the header h.

What are the back edges?

3->3 and 8->5

What are the natural loops?

highlighted in yellow above

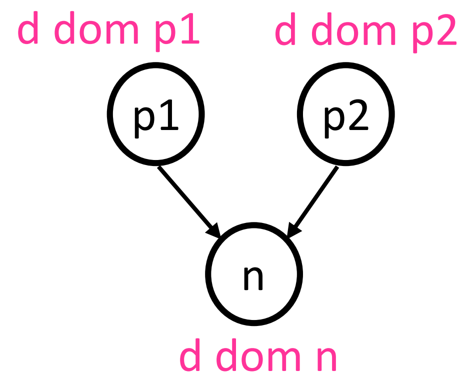
I. Algorithm to Find Natural Loops

Step 1. Find the dominator relations in a flow graph

Step 2. Identify the back edges

Step 3. Find the natural loop associated with the back edge

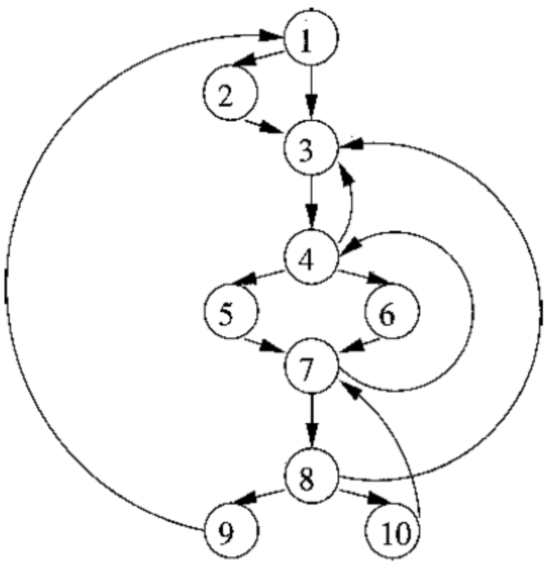
Step 1. Finding Dominators



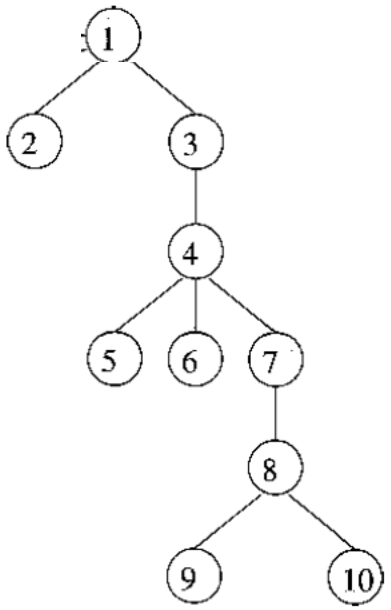
- Node *d* dominates node *n* in a graph (*d* dom *n*)
  if every path from the start node to *n* goes through *d*
- **Formulated as Data Flow Analysis problem:**
  - node *d* lies on all possible paths reaching node *n*
    iff d dom p for all pred p of n
    - Direction: forward
    - Values: basic blocks
    - Meet operator: ∩
    - Top (T): all basic blocks
    - Bottom: {}
    - Boundary condition for entry node: OUT[entry]= {entry}
    - Initialization for internal nodes OUT[b]= T
    - Finite descending chain? Yes (depth=number of basic blocks)
    - Transfer function: $f_b(x) = \{b\} \cup x$
    - Monotone & Distributive? Yes and yes: $(\{b\} \cup x) \cap (\{b\} \cup y) = \{b\} \cup (x \cap y)$
- **Speed:**
  - With rPostorder, most flow graphs (reducible flow graphs) converge in 1 pass

Example: Finding Dominators

OUT[b]={b} U (∩$_{\{p=pred(b)\}}$OUT[p])



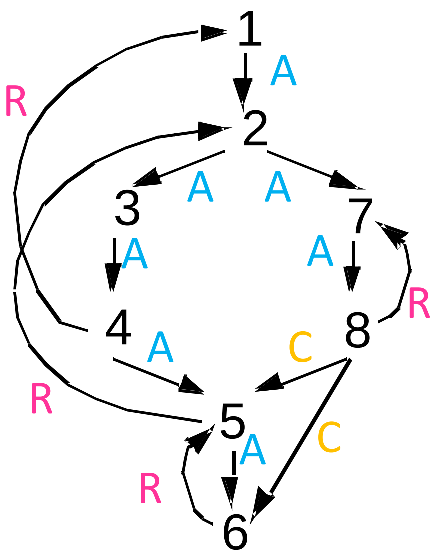
OUT[1] = {1}
OUT[2] = {1,2}
OUT[3] = {1,3}
OUT[4] = {1,3,4}
OUT[5] = {1,3,4,5}
OUT[6] = {1,3,4,6}
OUT[7] = {1,3,4,7}
OUT[8] = {1,3,4,7,8}
OUT[9] = {1,3,4,7,8,9}
OUT[10] = {1,3,4,7,8,10}
(No change in second iteration)



ALSU 9.6.1

Step 2. Finding Back Edges

- **Depth-first spanning tree**
  - Edges traversed in a depth-first search of the flow graph form a depth-first spanning tree

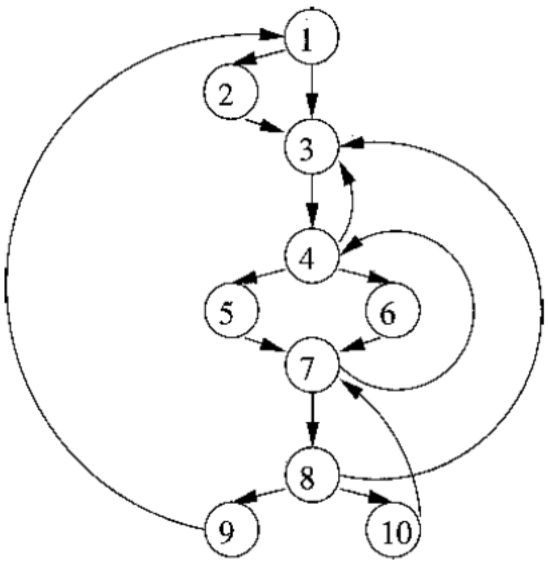


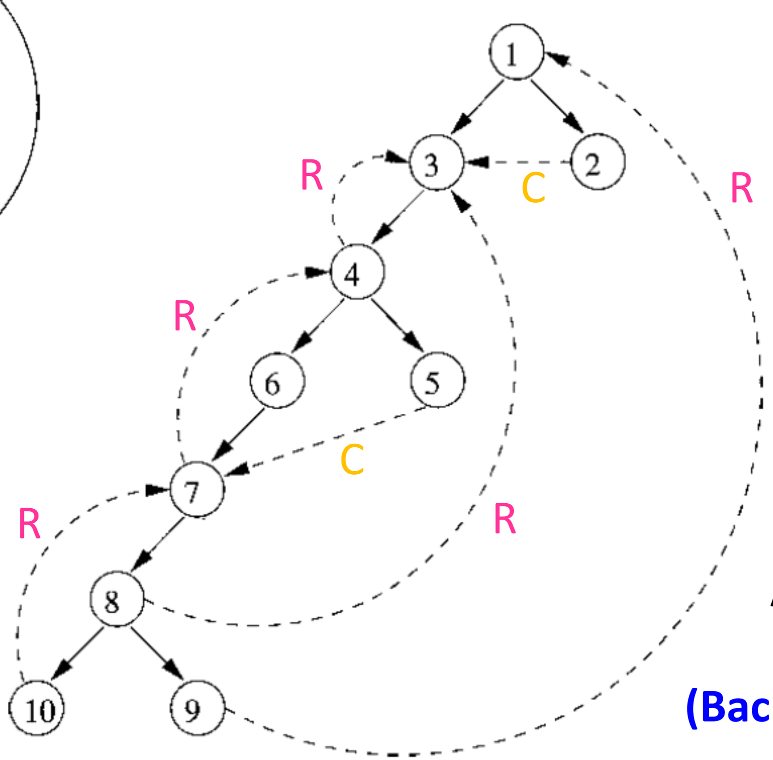
- **Categorizing edges in graph**
  - Advancing edges (A): from ancestor to proper descendant
  - Cross edges (C): from right to left
  - Retreating edges (R): from descendant to ancestor (not necessarily proper)

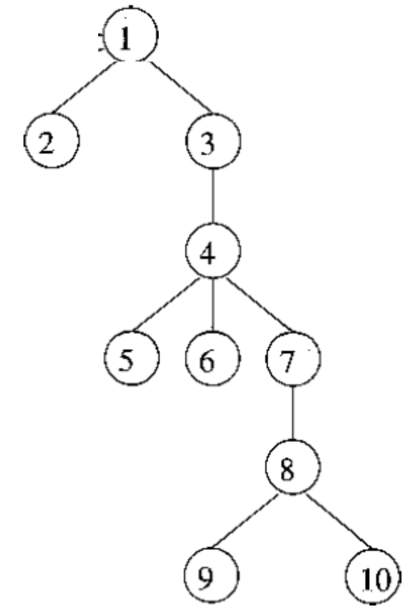
Back Edges

- **Definition**
  - **Back edge**: t->h, h dominates t
- **Relationships between graph edges and back edges**
- **Algorithm**
  - Perform a depth first search
  - For each retreating edge t->h, check if h is in t's dominator list
- **Most programs (all structured code, and most GOTO programs) have reducible flow graphs**
  - retreating edges = back edges

Example: Cross Edges, Retreating Edges, Back Edges







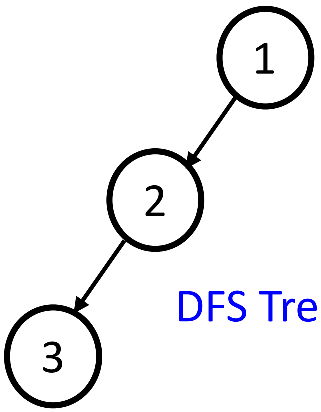
Which edges are:
cross edges?

retreating edges?

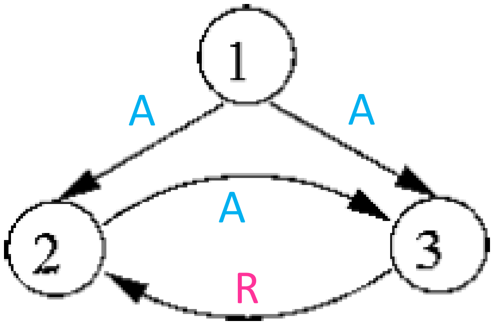
back edges?

All the retreating edges
are back edges
(**Back edge**: t->h, h dominates t)

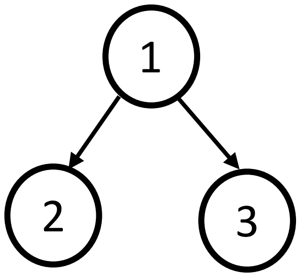
A Nonreducible Flow Graph



DFS Tree



not back edge

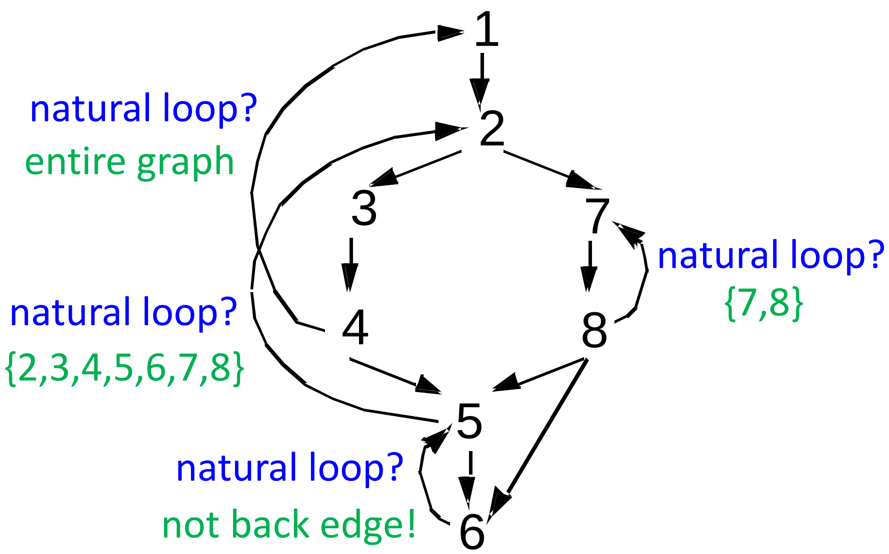


Dominator Tree

- **Categorizing edges in graph (relative to a DFS tree)**
  - Advancing edges (A): from ancestor to proper descendant
  - Cross edges (C): from right to left
  - Retreating edges (R): from descendant to ancestor (not necessarily proper)
    - Back edges: t->h, h dominates t

Step 3. Constructing Natural Loops

- The **natural loop of a back edge t->h** is the smallest set of nodes that includes t and h, and has no predecessors outside the set, except for the predecessors of the header h.

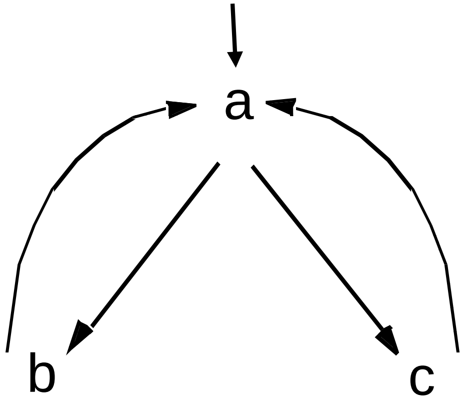


- **Algorithm: For each back edge t->h:**
  - delete *h* from the flow graph
  - find those nodes that can reach *t*
    (those nodes plus *h* form the natural loop of *t* -> *h*)

Inner Loops

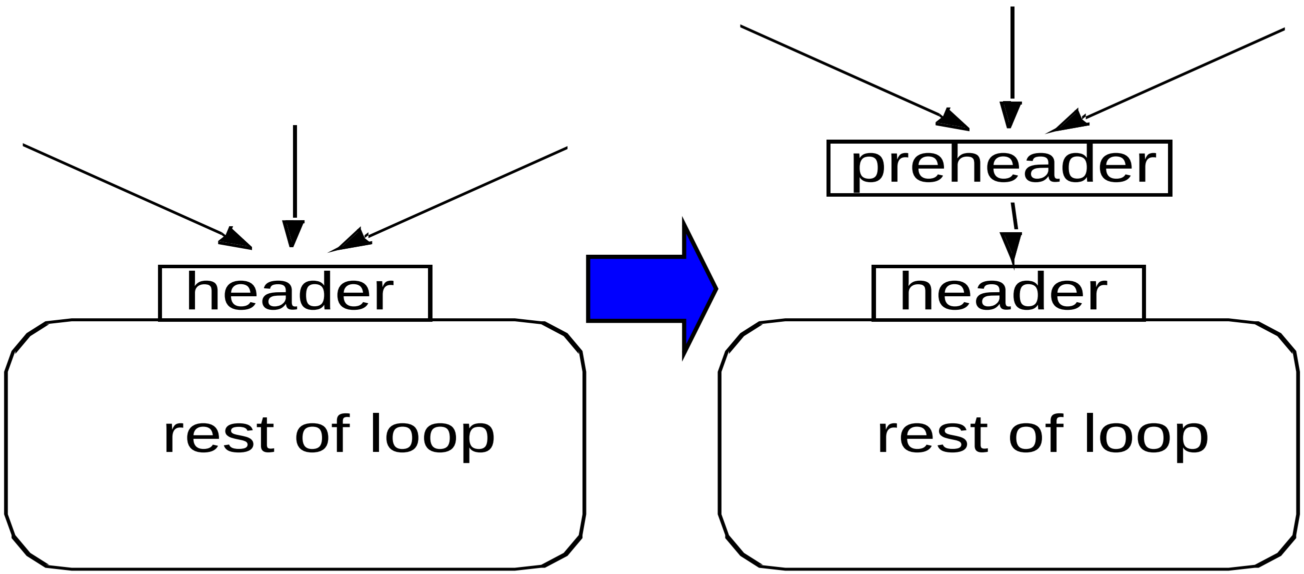
- **If two loops do not have the same header:**
  - they are either disjoint, or
  - one is entirely contained (nested within) the other
    - inner loop: one that contains no other loop.

- **If two loops share the same header:**
  - Hard to tell which is the inner loop
  - Solution: Combine and treat as one loop



Preheader

- **Optimizations often emit code that is to be executed once before the loop**
- **Solution: Create a preheader basic block for every loop**



Finding Loops: Summary

- **Define loops in graph theoretic terms**
- **Definitions and algorithms for:**
  - Dominators
  - Back edges
  - Natural loops

II. Overview of Induction Variable Elimination

Example

```
for(i=0; i<100; i++)
   A[i] = 0;
```

Induction variables:

t1 = 4i

t2 = 4i + &A

~~i = 0~~

L2: ~~IF i>=100~~ GOTO L1

~~t1 = 4 * i~~

~~t2 = &A + t1~~

*t2 = 0

~~i = i+1~~

GOTO L2

L1:

original code
(A[i] is 4 bytes)

```
t1′ = 0
t2′ = &A
IF t1′>=400
t1 = t1′
t2 = t2′

t1′ = t1′+4
t2′ = t2′+4
```

after induction variable substitution

```
    t2′ = &A
    t3′ = &A + 400
L2: IF t2′>=t3′ GOTO L1
    *t2′= 0
    t2′ = t2′+ 4
    GOTO L2
L1:
```

final code

```
    i = 0
L2: IF i>=100...
    t1 = 4 * i
    t2 = &A + t1
    *t2 = 0
    i = i+1
    GOTO L2
```

Definitions

- A **basic induction variable** is
  - a variable X whose only definitions within the loop are assignments of the form:

    X = X+c or X = X-c,

    where **c** is either a constant or a loop-invariant variable. (e.g., **`i`**)

- An **induction variable** is
  - a basic induction variable B, or
  - a variable defined once within the loop, whose value is a linear function of some basic induction variable at the time of the definition:

    $A = c_1 * B + c_2$ (e.g., **`t1`**, **`t2`**)

- The **FAMILY of a basic induction variable B** is
  - the set of induction variables A such that each time A is assigned in the loop, the value of A is a linear function of B. (e.g., **`t1`**, **`t2`** is in family of **`i`**)

Optimizations

1. **Strength reduction:**
   - A is an induction variable in family of basic induction variable B (i.e., $A = c_1 * B + c_2$)
     - Create new variable: A'
     - Initialize in preheader: $A' = c_1 * B + c_2$
     - Track value of B: add after B=B+x: $A'=A'+x*c_1$
     - Replace assignment to A: replace lone A=… with A=A'

```
     i = 0
L2:  IF i>=100 GOTO L1
     ~~t1 = 4 * i~~
     ~~t2 = &A + t1~~
     *t2 = 0
     i = i+1

     GOTO L2
```

```
t1' = 0
t2' = &A

t1 = t1'
t2 = t2'

t1' = t1'+4
t2' = t2'+4
```

Induction variables:

t1 = 4*i

t2 = 4*i + &A

Optimizations (continued)

2. Optimizing non-basic induction variables

- copy propagation
- dead code elimination

3. Optimizing basic induction variables

- Eliminate basic induction variables used only for
  - calculating other induction variables and loop tests
- Algorithm:
  - Select an induction variable A in the family of B, preferably with simple constants ($A = c_1 * B + c_2$).
  - Replace a comparison such as

```
if B > X goto L1
```

with

```
if (A' > c1*X + c2) goto L1
```

(assuming c_1 is positive)

  - if B is live at any exit from the loop, recompute it from A'
    - After the exit, $B = (A' - c_2) / c_1$

Carnegie Mellon

Example Continued

```
for(i=0; i<100; i++)
    A[i] = 0;
```

Induction variables:

t1 = 4i

t2 = 4i + &A

~~i = 0~~
L2: ~~IF i>=100~~ GOTO L1
~~t1 = 4 * i~~
~~t2 = &A + t1~~
~~*t2 = 0~~
~~i = i+1~~
GOTO L2
L1:

~~t1′ = 0~~
t2′ = &A
IF t2′>=&A+400
~~t1 = t1′~~
~~t2 = t2′~~
*t2′= 0
~~t1′ = t1′+4~~
t2′ = t2′+4

```
    t2′ = &A
    t3′ = &A + 400
L2: IF t2′>=t3′ GOTO L1
    *t2′= 0
    t2′ = t2′+ 4
    GOTO L2
L1:
```

$B >= X \Rightarrow A' >= c_1 * X + c_2$

III. Further Details

- **A BASIC induction variable in a loop L**
  - a variable X whose only definitions within L are assignments of the form:
    X = X+c or X = X-c, where c is either a constant or a loop-invariant variable.
- **Algorithm: can be detected by scanning L**
- Example:

```
k = 0;
for (i = 0; i < n; i++) {
    k = k + 3;
    … = m;
    if (x < y)
        k = k + 4;
    if (a < b)
        m = 2 * k;
    k = k – 2;
    … = m;
}
```

Basic induction variable(s)? **`i,k`**

Additional induction variable(s)?
`m = 2k+0` (in family of **`k`**)

Each iteration may execute a different number of increments/decrements!!

Strength Reduction Algorithm

- **Key idea**:
  - For each induction variable A, (A = c_1*B+c_2 at time of definition)
    - variable A' holds expression c_1*B+c_2 at all times
    - replace definition of A with A=A' only when executed

(`m` is only updated when appropriate)

- **Result**:
  - Program is correct
  - Definition of A does not need to refer to B

Finding Induction Variable Families

- **Let B be a basic induction variable**
  - Find all induction variables A in family of B:
    - $A = c_1 * B + c_2$
      (where B refers to the value of B at time of definition)
- **Conditions:**
  - C1: If A has a single assignment in the loop L, and assignment is one of:

```
A = B * c
A = c * B      (e.g., m)
A = B / c     (assuming A is real)
A = B + c
A = c + B
A = B – c
A = c – B
```

  - OR, ... (next page)

Finding Induction Variable Families (continued)

– C2: Let D be an induction variable in the family of B ($D = c_1 * B + c_2$)

Rule 1: If A has a single assignment in the loop L, and assignment is one of:

```
A = D * c
A = c * D
A = D / c    (assuming A is real)
A = D + c
A = c + D
A = D - c
A = c - D
```

Rule 2: No definition of D outside L reaches the assignment to A

Rule 3: Every path between the lone point of assignment to D in L and the assignment to A has the same sequence (possibly empty) of definitions of B

Induction Variable Family Example 1

```
L2: IF i>=100 GOTO L1
    t2 = t1 + 10
    t1 = 4 * i
    t3 = t1 * 8
    i = i + 1
    goto L2
L1:
```

Is **i** a basic induction variable? yes

Is **t2** in family of **i**? no (fails Rule 2)

Is **t1** in family of **i**? yes (by C1)

Is **t3** in family of **i**? yes (by C2 with A:**t3**, D:**t1**, B:**i**)

Condition C1

A has a single assignment in the loop L of the form A = B*c, c*B, B+c, etc

Condition C2

A is in family of B if D = c_1* B + c_2 for basic induction variable B and:

- Rule 1: A has a single assignment in the loop L of the form A = D*c, D+c, etc
- Rule 2: No definition of D outside L reaches the assignment to A
- Rule 3: Every path between the lone point of assignment to D in L and the assignment to A has the same sequence (possibly empty) of definitions of B

Induction Variable Family Example 2

```
L3: IF i>=100 GOTO L1
    t1 = 4 * i
    IF t1 < 50 GOTO L2
    i = i + 2
L2: t2 = t1 + 10
    i = i + 1
    goto L3
L1:
```

Is **i** a basic induction variable? yes (all are **i=i+c**)

Is **t1** in family of **i**? yes (by C1)

Is **t2** in family of **i**? no (fails Rule 3)

Condition C1

A has a single assignment in the loop L of the form A = B*c, c*B, B+c, etc

Condition C2

A is in family of B if D = c_1* B + c_2 for basic induction variable B and:

- Rule 1: A has a single assignment in the loop L of the form A = D*c, D+c, etc
- Rule 2: No definition of D outside L reaches the assignment to A
- Rule 3: Every path between the lone point of assignment to D in L and the assignment to A has the same sequence (possibly empty) of definitions of B

Induction Variables Summary

- **Precise definitions of induction variables**
- **Systematic identification of induction variables**
- **Strength reduction**
- **Clean up:**
  - eliminating basic induction variables
    - used in other induction variable calculations
    - replacement of loop tests
  - eliminating other induction variables
    - standard optimizations

Today's Class

I. Finding loops

II. Overview of Induction Variable Optimizations

III. Further details

Wednesday's Class

Loop Invariant Code Motion ALSU 9.5-9.5.2