U.PORTO FEUP FACULDADE DE ENGENHARIA UNIVERSIDADE DO PORTO

DEI DEPARTAMENTO DE ENGENHARIA INFORMÁTICA

# Compilers Design and Implementation

## *Loop Optimizations*

Loop Invariant Code Motion
Induction Variables

# Outline

- Loop Invariant Code Motion
- Induction Variables Recognition

# Loop Invariant Code Motion

- If a computation produces the same value in every loop iteration, move it out of the loop

# Loop Invariant Code Motion

- If a computation produces the same value in every loop iteration, move it out of the loop

```
for i = 1 to N
  x = x + 1
  for j = 1 to N
     a(i,j) = 100*N + 10*i + j + x
```

# Loop Invariant Code Motion

- If a computation produces the same value in every loop iteration, move it out of the loop

```
for i = 1 to N
  x = x + 1
  for j = 1 to N
     a(i,j) = 100*N + 10*i + j + x
```

# Loop Invariant Code Motion

- If a computation produces the same value in every loop iteration, move it out of the loop

```
t1 = 100*N
for i = 1 to N
  x = x + 1
  for j = 1 to N
     a(i,j) = 100*N + 10*i + j + x
```

U.PORTO FEUP FACULDADE DE ENGENHARIA UNIVERSIDADE DO PORTO
DEI DEPARTAMENTO DE ENGENHARIA INFORMÁTICA

# Loop Invariant Code Motion

- If a computation produces the same value in every loop iteration, move it out of the loop

```
t1 = 100*N
for i = 1 to N
  x = x + 1
  for j = 1 to N
     a(i,j) = t1 + 10*i + j + x
```

# Loop Invariant Code Motion

- If a computation produces the same value in every loop iteration, move it out of the loop

```
t1 = 100*N
for i = 1 to N
  x = x + 1
  for j = 1 to N
     a(i,j) = t1 + 10*i + j + x
```

# Loop Invariant Code Motion

- If a computation produces the same value in every loop iteration, move it out of the loop

```
t1 = 100*N
for i = 1 to N
  x = x + 1
  for j = 1 to N
     a(i,j) = t1 + 10*i + j + x
```

# Loop Invariant Code Motion

- If a computation produces the same value in every loop iteration, move it out of the loop

```
t1 = 100*N
for i = 1 to N
  x = x + 1
  t2 = 10*i + x
  for j = 1 to N
     a(i,j) = t1 + 10*i + j + x
```



DEI DEPARTAMENTO DE ENGENHARIA INFORMÁTICA
U.PORTO FEUP FACULDADE DE ENGENHARIA UNIVERSIDADE DO PORTO

# Loop Invariant Code Motion

- If a computation produces the same value in every loop iteration, move it out of the loop

```
t1 = 100*N
for i = 1 to N
  x = x + 1
  t2 = 10*i + x
  for j = 1 to N
     a(i,j) = t1 + t2 + j
```

DEI DEPARTAMENTO DE ENGENHARIA INFORMÁTICA
U.PORTO FEUP FACULDADE DE ENGENHARIA UNIVERSIDADE DO PORTO

# Loop Invariant Code Motion

- If a computation produces the same value in every loop iteration, move it out of the loop

```
t1 = 100*N
for i = 1 to N
  x = x + 1
  t2 = 10*i + x
  for j = 1 to N
     a(i,j) = t1 + t2 + j
```

# Loop Invariant Code Motion

- If a computation produces the same value in every loop iteration, move it out of the loop

```
t1 = 100*N
for i = 1 to N
  x = x + 1
  t2 = 10*i + x
  for j = 1 to N
     a(i,j) = t1 + t2 + j
```

- Correctness and Profitability?
  - Loop Should Execute at Least Once!

# Opportunities for LICM

- In User Code
  - Complex Expressions
  - Easily readable code, reduce # of variables

- After Compiler Optimizations
  - Copy Propagation, Algebraic simplification

# Usefulness of LICM

- Many programs Spend Most of their execution time in loops
- Reducing work inside a loop nest is very beneficial
  - CSE of expression ⇒ x instructions become x/2
  - LICM of expression ⇒ x instructions become x/N

# Implementing LICM

- If a computation produces the same value in every loop iteration, move it out of the loop
- An expression can be moved out of the loop if all its operands are invariant in the loop

# Invariant Operands

- Constant Values
- Variables whose definitions are outside the loop

# Invariant Operands

- Constant Values
- Variables whose definitions are outside the loop

```
x = f(...)
y = g(...)
for i = 1 to N
    t = t + x*y
```

DEI DEPARTAMENTO DE ENGENHARIA INFORMÁTICA
U.PORTO FEUP FACULDADE DE ENGENHARIA UNIVERSIDADE DO PORTO

# Invariant Operands

- Constant Values
- Variables whose definitions are outside the loop
- Operand has only one reaching definition *and* that definition is loop invariant

# Invariant Operands

- Constant Values
- Variables whose definitions are outside the loop
- Operand has only one reaching definition *and* that definition is loop invariant

```
for i = 1 to N
    x = 100
    y = x * 5
```

# Invariant Operands

- Constant Values
- Variables whose definitions are outside the loop
- Operand has only one reaching definition *and* that definition is loop invariant

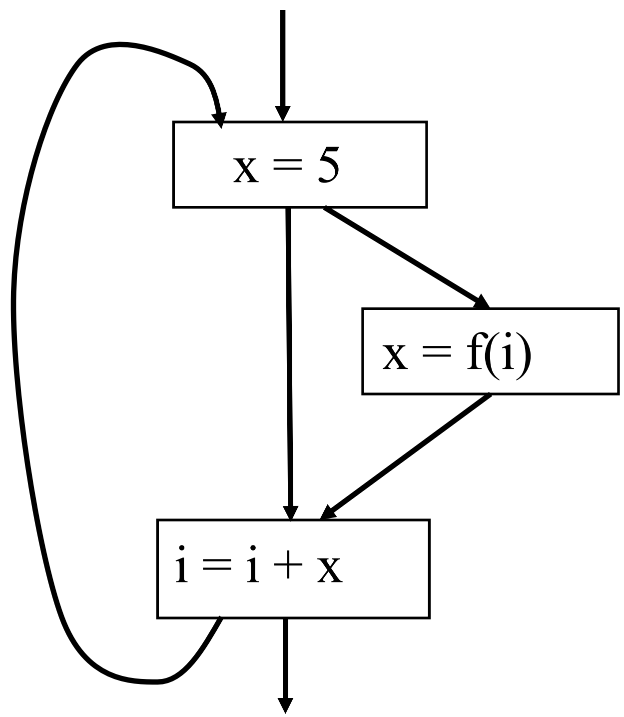
```
for i = 1 to N
  x = 100
  y = x * 5
```

```
for i = 1 to N
 if i > p then
   x = 10
 else
   x = 5
 y = x * 5
```

- Clearly a single definition is a safe restrictions
  - There could be many definition with the same value

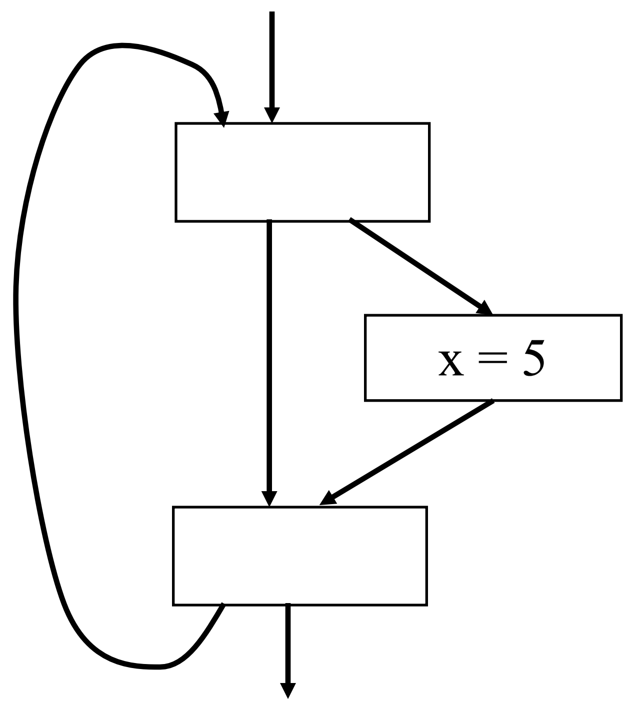
# Move Or Not To Move….

- Statement can be moved only if
  - all the Uses are Dominated by the Statement



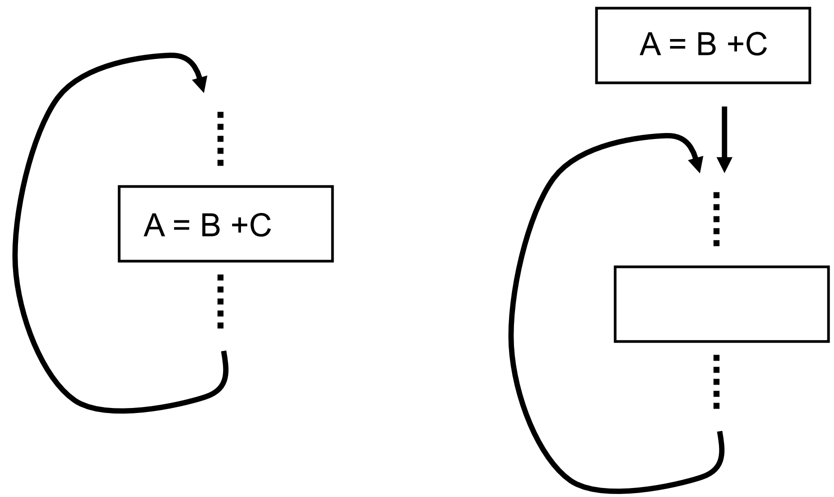
# Move Or Not To Move….

- Statement can be moved only if
  - All the Uses are Dominated by the Statement
  - The Exit of the Loop is Dominated by the Statement



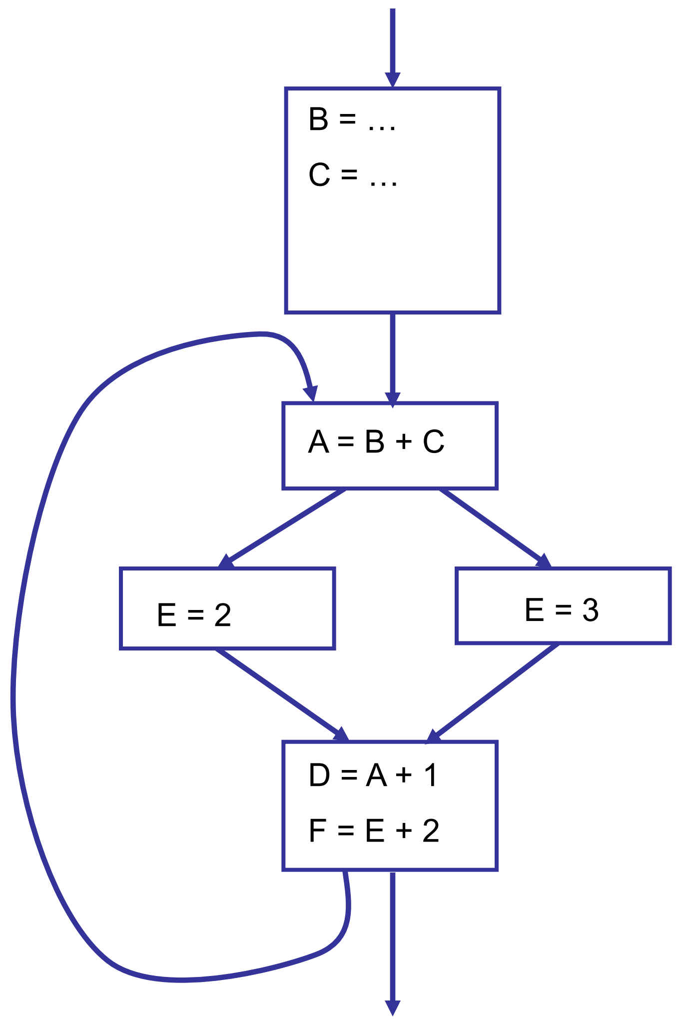
# Conditions for Code Motion

- Correctness: Movement doesn't change the semantics of the program
- Performance: Code is not slowed down

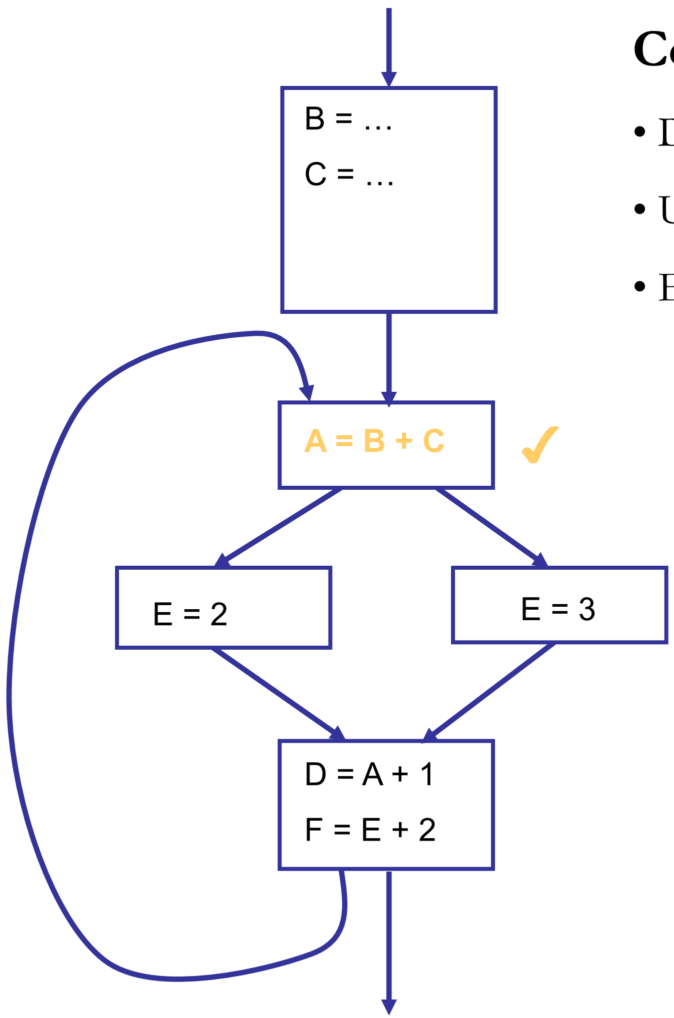



- Basic Ideas - Defines once and for all
  - Control flow
  - Other definitions
  - Other uses

# Example: LICM



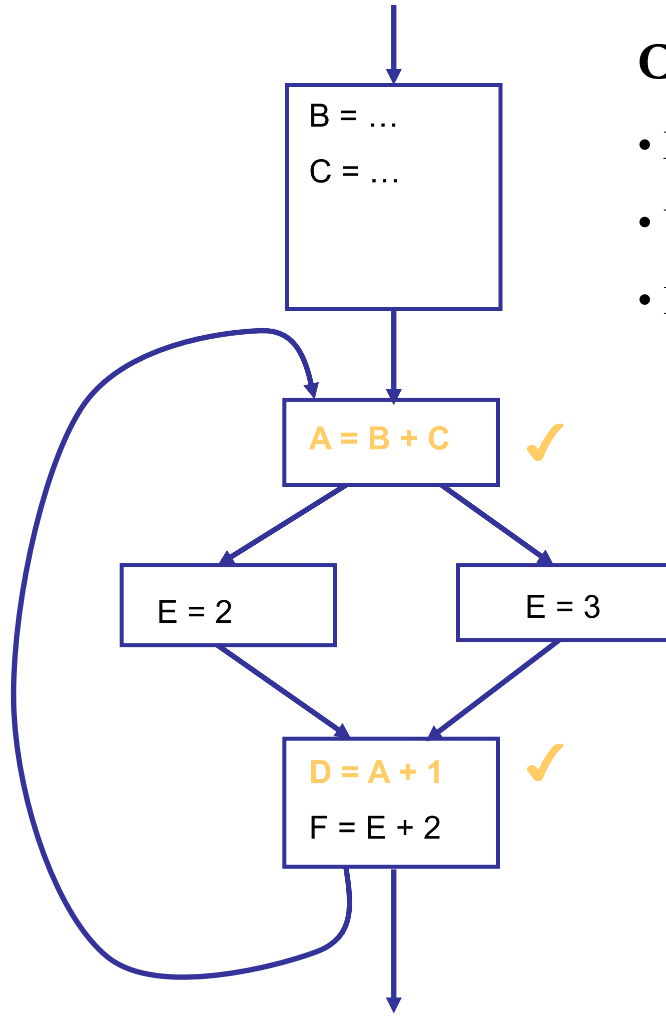
# Example: LICM




**Conditions:**

- Defs. of B and C outside the Loop
- Uses of A dominated by Statement
- Exit Dominated by Statement

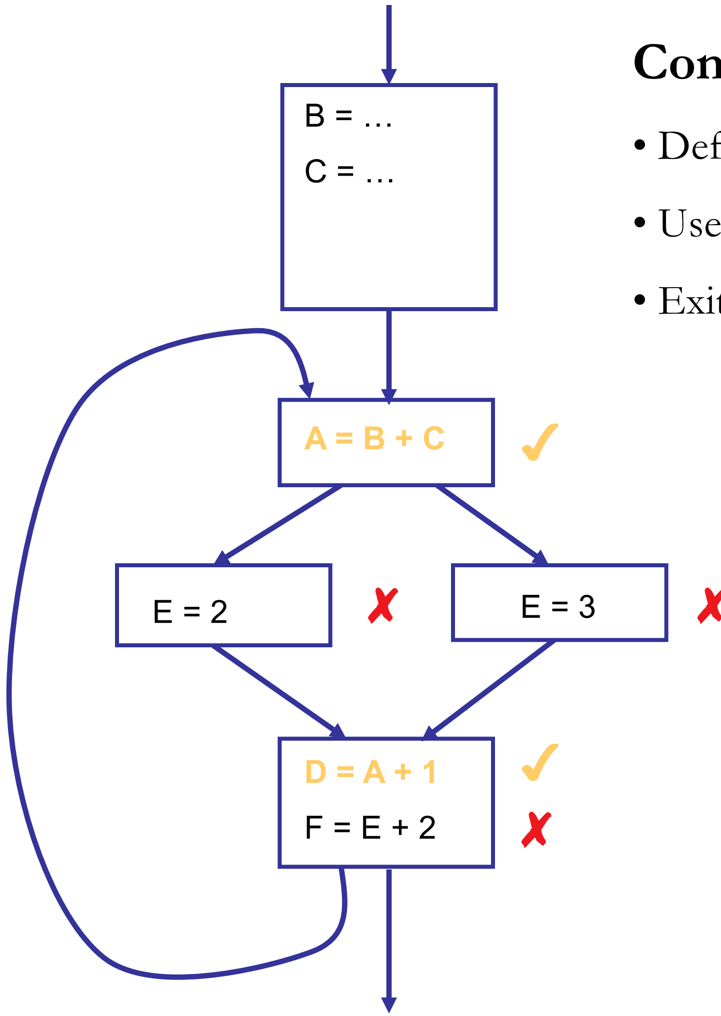
# Example: LICM




## Conditions:

- Defs. of B and C outside the Loop
- Uses of A dominated by Statement
- Exit Dominated by Statement

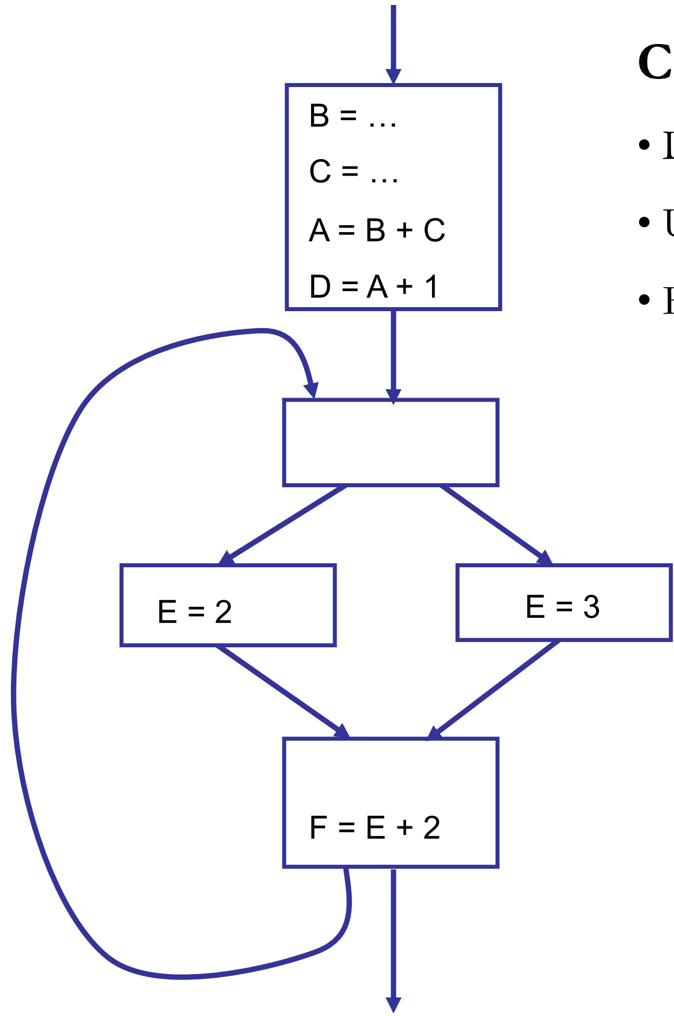
# Example: LICM




**Conditions:**

- Defs. of B and C outside the Loop
- Uses of A dominated by Statement
- Exit Dominated by Statement

# Example: LICM




**Conditions:**

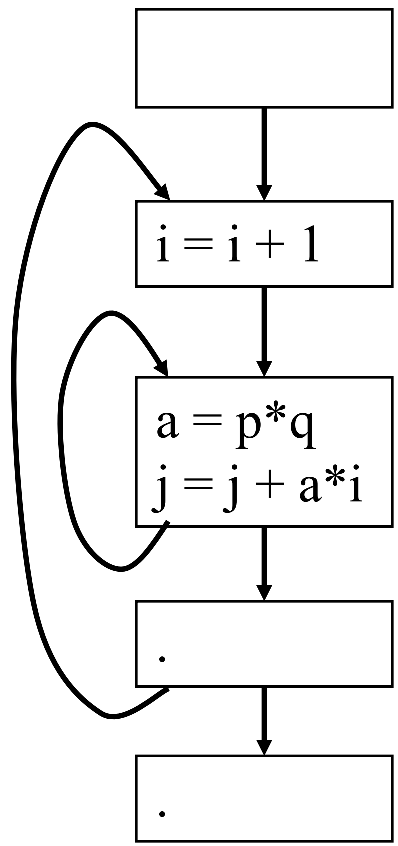
- Defs. of B and C outside the Loop
- Uses of A dominated by Statement
- Exit Dominated by Statement

# Handling Nested Loops

- Process Loops from Innermost to Outermost

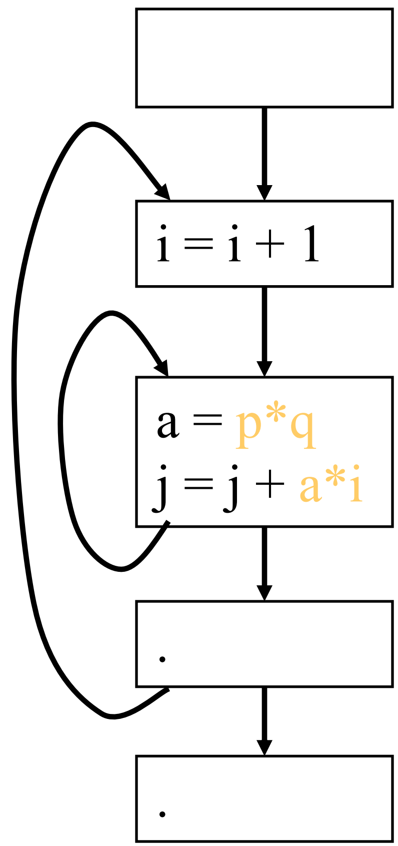
# Handling Nested Loops

- Process loops from innermost to outermost



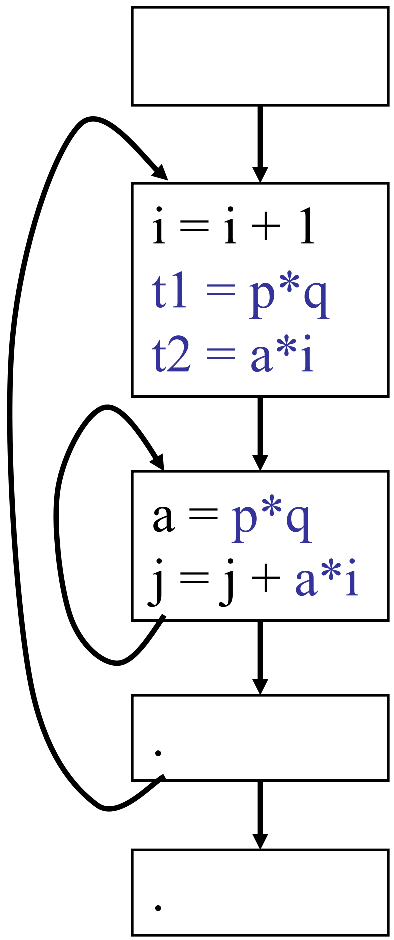
# Handling Nested Loops

- Process loops from innermost to outermost



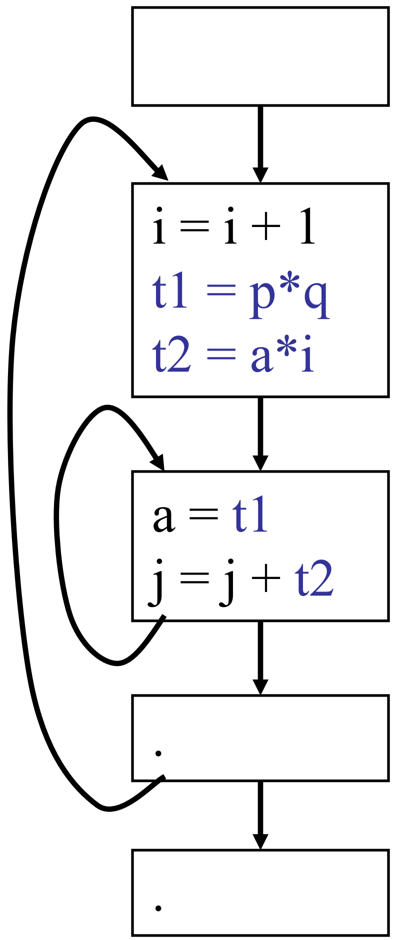
# Handling Nested Loops

- Process loops from innermost to outermost



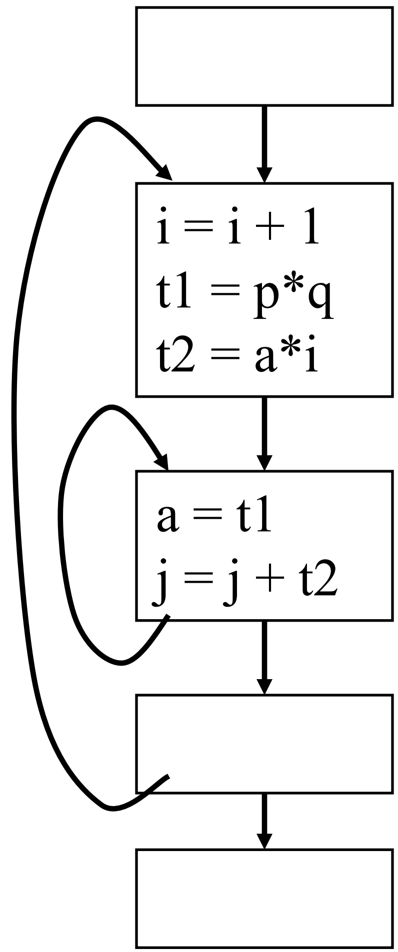
# Handling Nested Loops

- Process loops from innermost to outermost



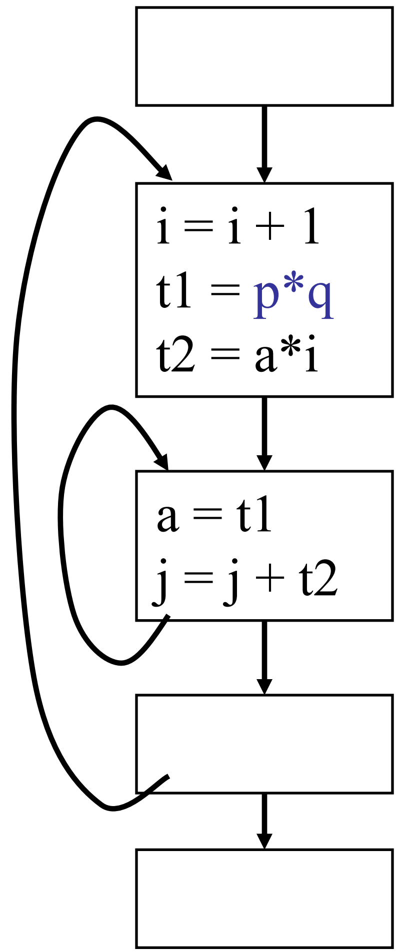
# Handling Nested Loops

- Process loops from innermost to outermost



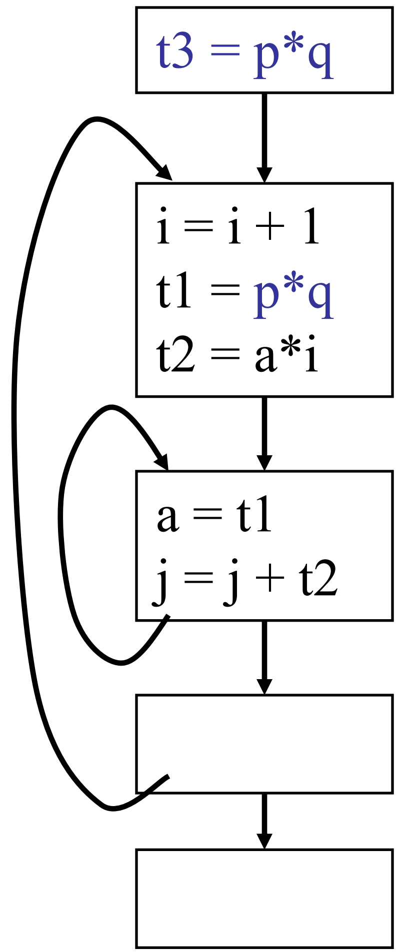
# Handling Nested Loops

- Process loops from innermost to outermost



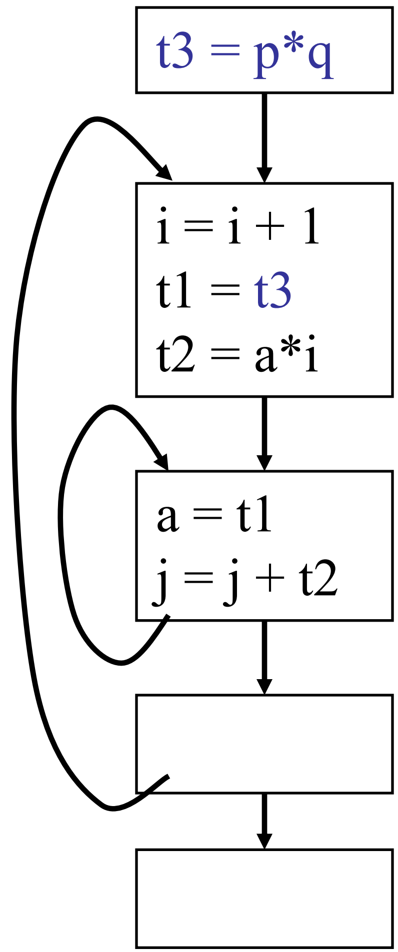
# Handling Nested Loops

- Process loops from innermost to outermost



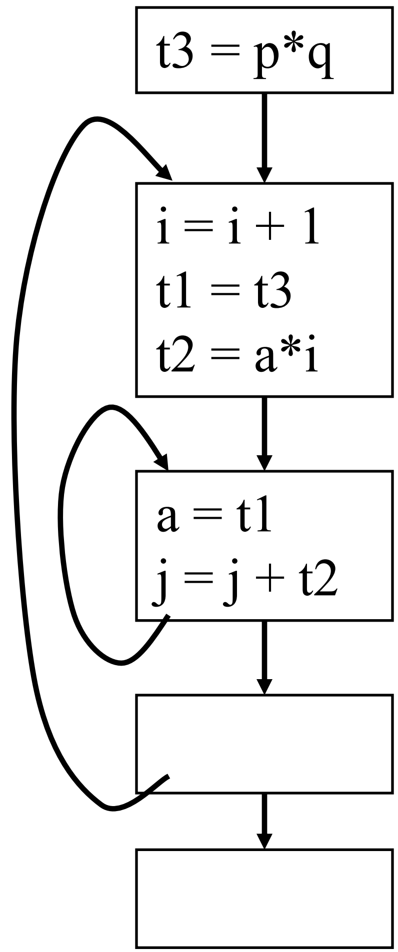
# Handling Nested Loops

- Process loops from innermost to outermost



# Handling Nested Loops

- Process loops from innermost to outermost



# Algorithm for LICM

- Observations
  - Loop Invariant
    - Operands are defined outside loop or invariant themselves
  - Code Motion
    - Not all loop invariant instructions can be moved to pre-header.
    - Why?

- **Algorithm**
  - Find Invariant Expression
  - Check Conditions for Code Motion
  - Apply Code Transformation

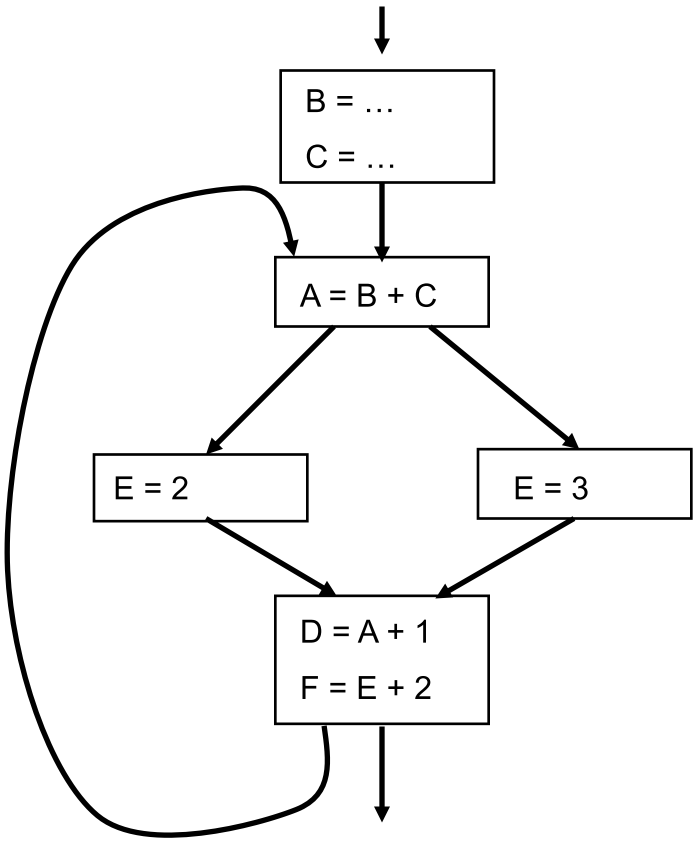
# Detecting Loop Invariant Computation

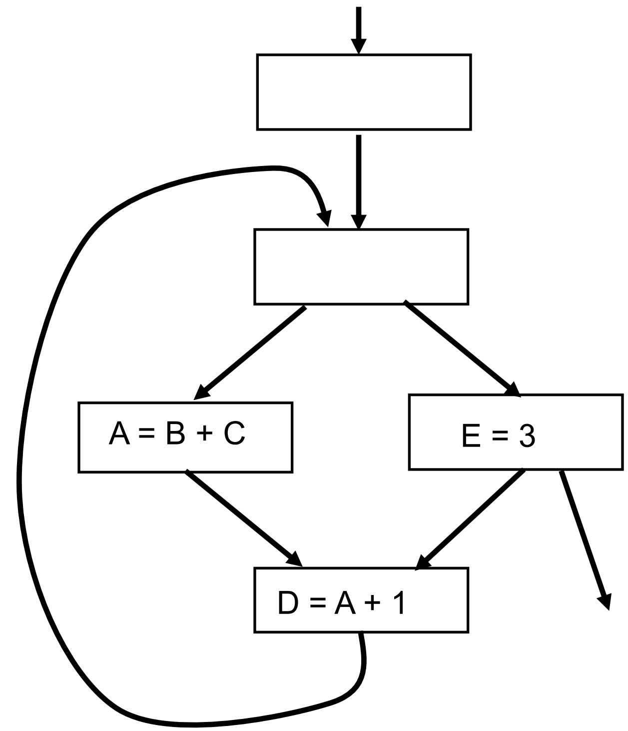
- Algorithm
  1. Compute Reaching Definitions for every variable in every Basic Block
  2. Mark Invariant a statement `s: a = b+c` if
     - All definitions of `b` and `c` that reach the statement `s` are outside the loop
       - What about constants `b`, `c` ?
  3. Repeat: Mark Invariant if
     - All reaching definitions of `b` are outside the loop, or
     - There is exactly one reaching definition for `b`, and it is from a loop-invariant statement inside the loop
     - Idem for `c`
     - Until no changes to set of loop-invariant statements.

# Code Motion Algorithm

- Given: a set of nodes in a loop
  - Compute Reaching Definitions
  - Compute Loop Invariant Computation
  - Compute Dominators
  - Find the exits of the loop, nodes with successors outside the loop
  - Candidate Statement for Code Motion:
    - Loop Invariant
    - In blocks that dominate all the Exits of the Loop
    - Assign to variable not assigned to elsewhere in the loop
    - In blocks that dominate all blocks in the loop that use the variable assigned
  - Perform a depth-first search of the blocks
    - Move candidate to pre-header if all the invariant operations it depends on have been moved

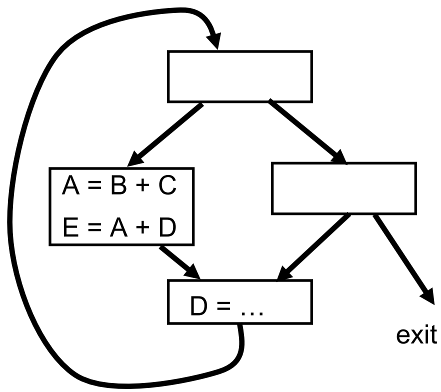
# Examples



# More Aggressive Optimizations

- Gamble On: Most loops get executed
  - Can we relax the constraint of dominating all exits?




- Landing Pads

```
While p do s    ⇒        if p {
                              pre-header
                              repeat
                                 statements
                              until not p;
                         }
```

# Summary

- Loop Invariant Code Motion
  - Important and Profitable Transformation
  - Precise Definition and Algorithm for Loop Invariant computation
  - Precise Algorithm for code motion
- Combination of Several Analyses
  - Use of Reaching Definitions (DU-chains)
  - Use Dominators
- Combination with Loop Induction Variables next

# Redundancy Elimination

- We did two optimizations
  - Common Sub-Expression Elimination
  - Loop Invariant Code Motion
  - Dead Code Elimination
- There are many others
  - Value Numbering
  - Partial redundancy elimination

# Induction Variables in Loops

- What is an Induction Variable?
  - For a given loop variable v is an induction variable iff
    - Its value Changes at Every Iteration
    - Is either incremented or decremented by a Constant Amount
      - Either Compile-time Known or Symbolically Constant…
- Classification:
  - Basic Induction Variables
    - A single assignment in the loop of the form x = x + constant
    - Example: variable i in for i = 1 to 10
  - Derived Induction Variables
    - A linear function of a basic induction variable
    - variable j in the loop assigned $j = c_1 * i + c_2$

# Why Are Induction Variables Important?

- Pervasive in Computations that Manipulate Arrays
  - Allow for Understanding of Data Access Patterns in Memory Access
    - Support Transformations Tailored to Memory Hierarchy
  - Can Be Eliminated with Strength Reduction
    - Substantially reduce the weight of address calculations
    - Combination with CSE

- Example:

```
for i = 1 to N
  for j = 1 to N
    a(i,j) = b(i,j)
```

```
for i = 1 to N
  t1 = @a(i,1)
  t2 = &b(i,1)
  for j = 1 to N
    *t1  = *t2
     t1 += 8
     t2 += 8
```

?

# Detection of Induction Variables

- Algorithm:
  - Inputs: Loop L with Reaching Definitions and Loop Invariant
  - Output: For each Induction Variable j the triple (i,c,d) s.t. the value of $j = i * c + d$
  - Find the Basic Induction Variables by Scanning the Loop L such that each Basic Induction Variable has (i,1,0)
  - Search for variables k with a single assignment to k of the form:
    - $k = j * b$, $k = b*j$, $k = j/b$, $k =+j$ with b a constant and j a basic induction variable
  - Check if the assignment dominates the definition points for j

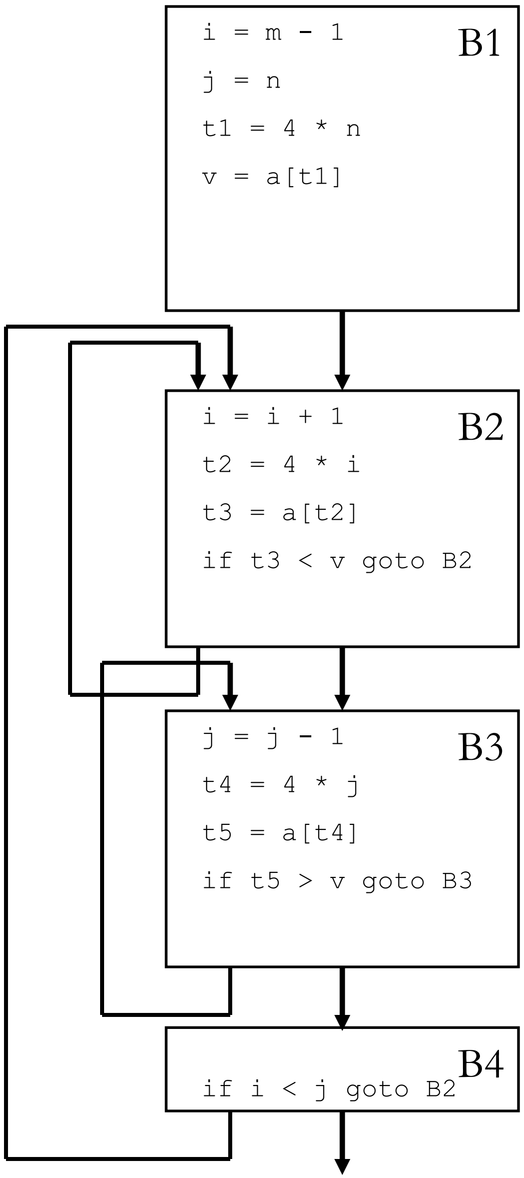
# Strength Reduction & Induction Variables

- Idea of the Transformation
  - Replace the Induction Variable in each Family by references to a common induction variable, the basic induction variable.
  - Exploit the Algebraic Properties for the update to the basic variable

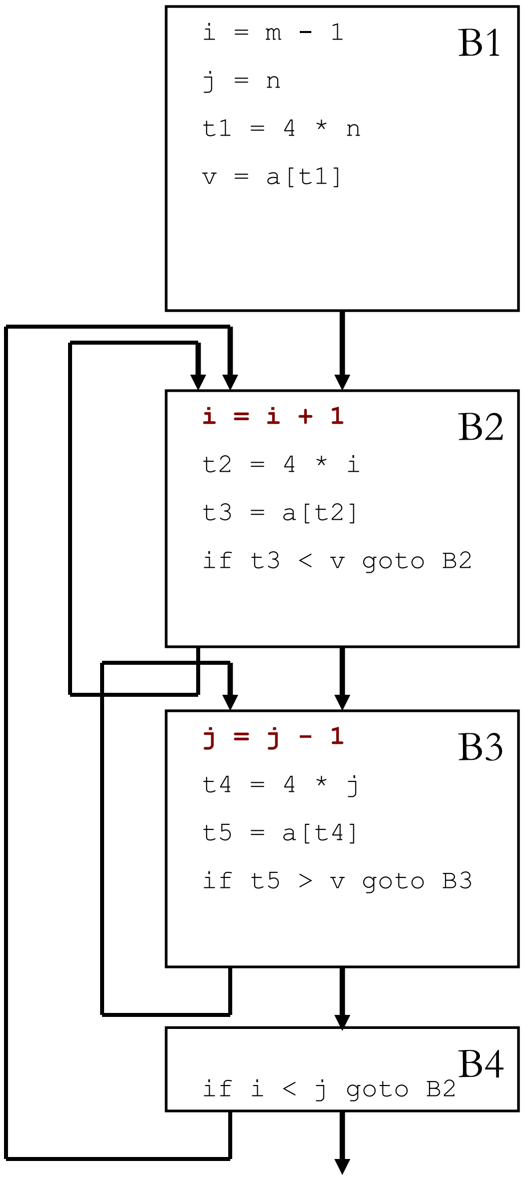
- Algorithm

```
foreach Basic Induction variable i do
     foreach Induction variable j: (i,c,d) in the family of i do
       create a new variable s
       replace the assignment to j by j = s
       after each assignment i = i + n where n is a constant
         append s = s + c * n
       place s in the family of induction variables of i
     end foreach
     initialize s to c*i + d on loop entry as
     either s = c * i followed by s = s + d (simplify if d = 0 or c = 1)
end foreach
```

# Detection of Induction Variables Example

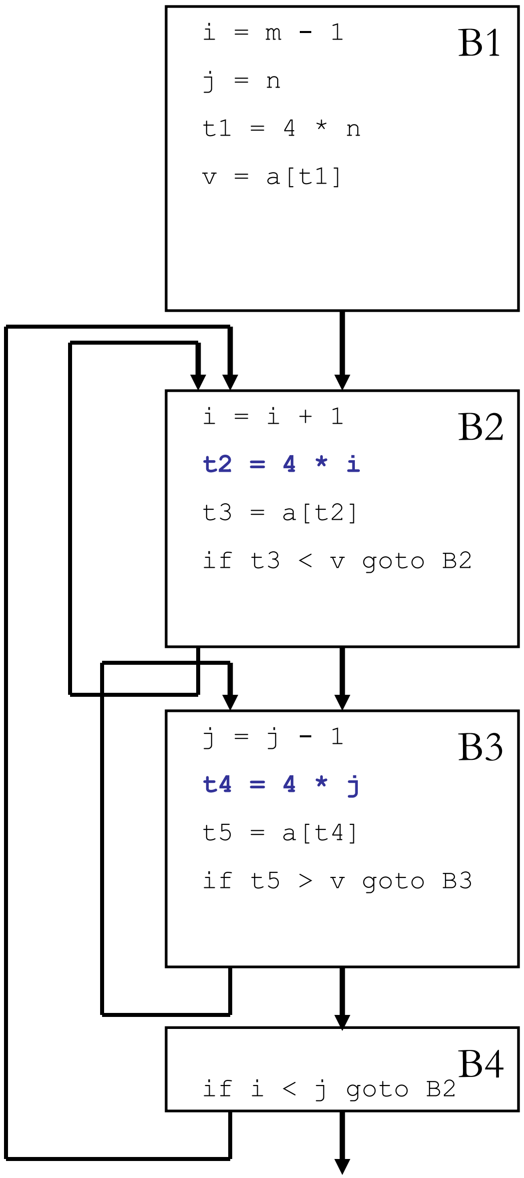


# Detection of Induction Variables Example



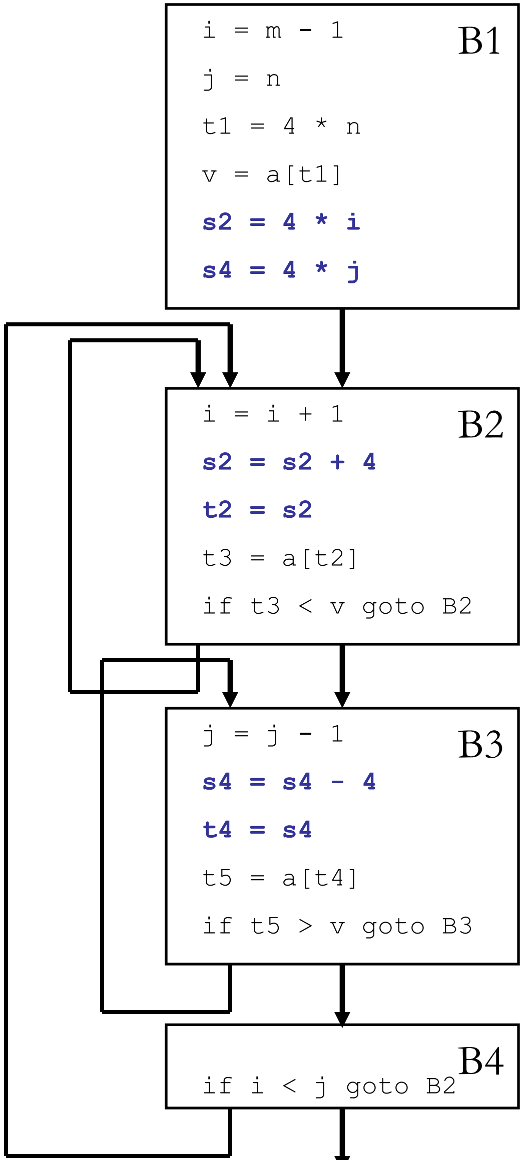

- Basic Induction Variables:
  - i in B2: single increment, (i,1,1)
  - j in B3: single decrement (j,1,-1)

# Detection of Induction Variables Example




- Basic Induction Variables:
  - i in B2: single increment, (i,1,1)
  - j in B3: single decrement (j,1,-1)
- Derived Induction Variables
  - t2 in B2: single assign (i,4,0)
  - t4 in B3: single assign (j,4,0)

# Strength Reduction of Induction Variables



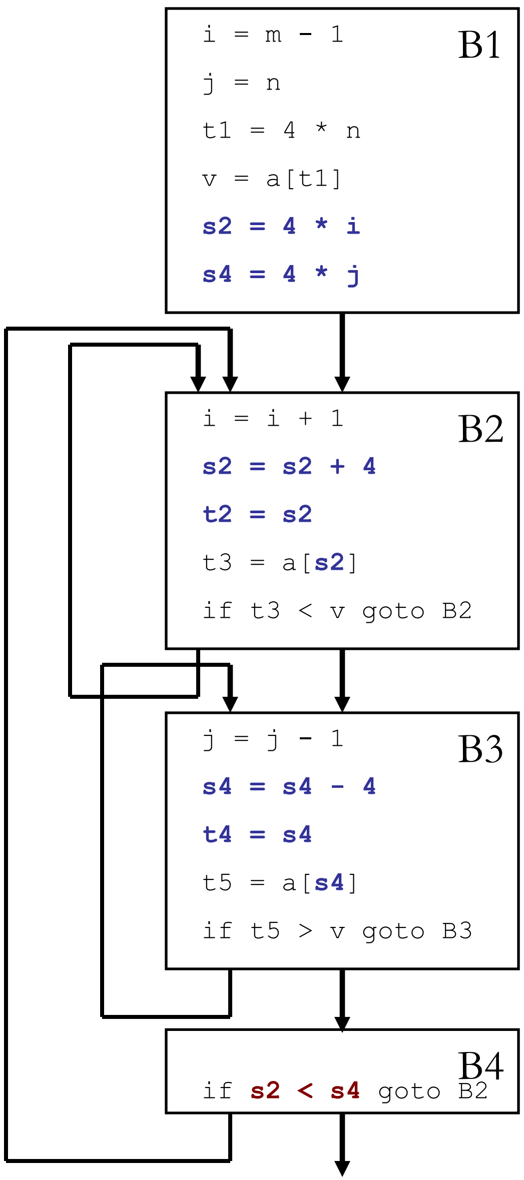

- Basic Induction Variables:
  - i in B2: single increment, (i,1,1)
  - j in B3: single decrement (j,1,-1)
- Derived Induction Variables
  - t2 in B2: single assign (i,4,0)
  - t4 in B3: single assign (j,4,0)
- Strength Reduction (for t4 in B3)
  - create s4 for the expression 4*j
  - replace t4 = 4*j with t4 = s4
  - replace induction step for j with s4 = s4 - 4 where -4 comes from -1*c
  - create initialization of s4 in pre-header

DEI DEPARTAMENTO DE ENGENHARIA INFORMÁTICA
U.PORTO FEUP FACULDADE DE ENGENHARIA UNIVERSIDADE DO PORTO

# Eliminating Induction Variables

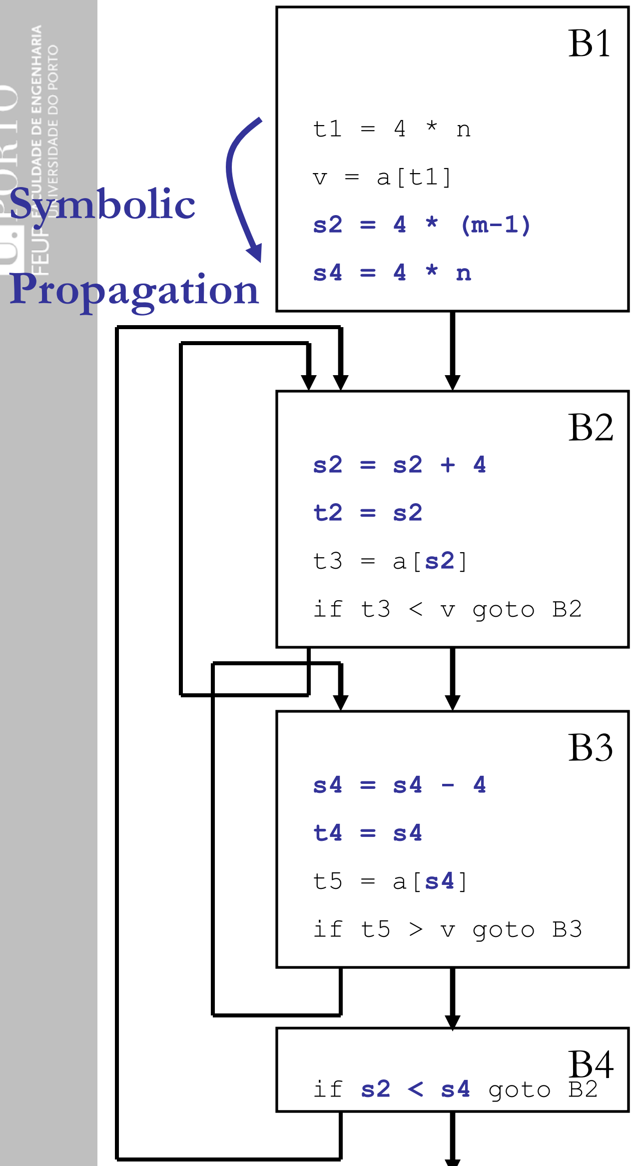
- After all the tricks we might be left with
  - Code that uses the basic induction variables just for conditional including the loop control
- Given the linear relation between induction variables
  - we can remove the basic induction variable by rewording the tests with a derived induction variable that is used in the code.
  - Check out dead statements (trivial is you use SSA)
  - Check the initialization and remove induction variables.

# Strength Reduction of Induction Variables




- Basic Induction Variables:
  - i in B2: single increment, (i,1,1)
  - j in B3: single decrement (j,1,-1)
- Derived Induction Variables
  - t2 in B2: single assign (i,4,0)
  - t4 in B3: single assign (j,4,0)
- Strength Reduction (for t4 in B3)
  - create s4 for the expression 4*j
  - replace t4 = 4*j with t4 = s4
  - replace induction step for j with s4 = s4 - 4 where -4 comes from -1*c
  - create initialization of s4 in pre-header
- Elimination of Induction Variables
  - replace i < j with s2 < s4
  - copy propagate s2 and s4

# Strength Reduction of Induction Variables




- Basic Induction Variables:
  - i in B2: single increment, (i,1,1)
  - j in B3: single decrement (j,1,-1)
- Derived Induction Variables
  - t2 in B2: single assign (i,4,0)
  - t4 in B3: single assign (j,4,0)
- Strength Reduction (for t4 in B3)
  - create s4 for the expression 4*j
  - replace t4 = 4*j with t4 = s4
  - replace induction step for j with s4 = s4 - 4 where -4 comes from -1*c
  - create initialization of s4 in pre-header
- Elimination of Induction Variables
  - replace i < j with s2 < s4
  - copy propagate s2 and s4

# Summary

- Induction Variables
  - Change Values at Every Iteration of a Loop by a Constant amount
  - Basic and Derived Induction Variables with Affine Relation
- Great Opportunity for Transformations
  - Pervasive in Loops that Manipulation Array Variables
  - Loop Control and Array Indexing
- Combination of Various Analyses and Transformations
  - Dominators, Reaching Definitions
  - Strength Reduction, Dead Code Elimination and Copy Propagation and Common Sub-Expression Elimination