SSA form in LLVM IR
and
Control flow analysis

Nikhil Hegde

Compiler Optimizations course @ QUALCOMM India Pvt. Ltd.

Content

- SSA
- CFGs
- Dominator trees and Dominance frontiers
- Application of concepts: loop invariant code motion

Single assignment form - motivation

x=z+y
a=x
x=2*x

replace x with b

b=z+y
a=b
x=2*b

Single assignment form: a variable is assigned only once i.e. appears only once in LHS.

Aids CSE:

```
x=z+y
. . .
x=z+y
```

Neither z nor y can appear on the LHS here in single assignment form.

So, can be sure that this z+y is the same expression as earlier. In the original code, if z or y were assigned to in between the two expressions, then we would have used different names, say, z1=..; y1=; then the last expression would have to be rewritten as x=z1+y1.

Aids copy propagation: can replace all the uses of a variable downstream

Aids dead code elimination: if the variable is never used later, can safely remove the statement where the variable is defined/assigned to.

SSA

- When there is control flow, tricky to get SSA form.

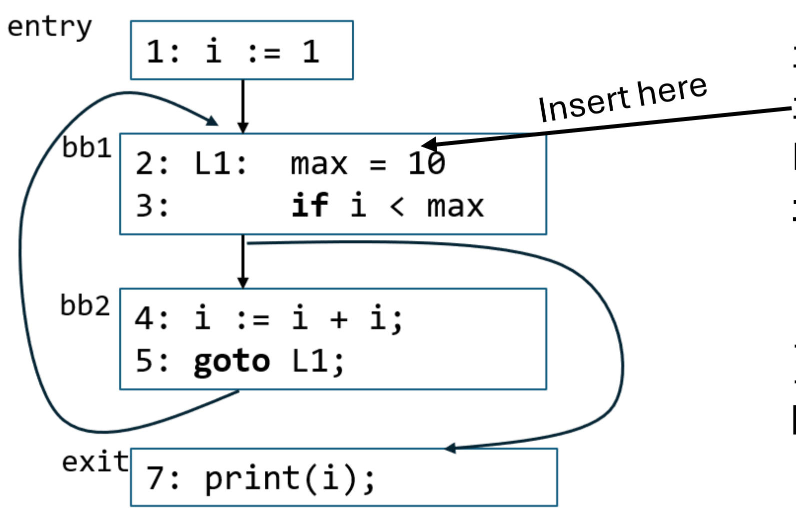
```
1:        i1 = 1
2: L1:    max = 10
3:        if i1 < max {
4:         i2 = i1 + i1
5:         goto L1;
6:        }
7:        print(i);
```

`i1 or i2 ?`

Phi nodes – special instructions that help deal with control flow. E.g.,
`i3 = phi [i1, bb1], [i2, bb4]`

- **Exercise:** draw the corresponding CFG.

Insert phi node




```
i1 := 1
i2 = phi [i1, entry], [i3, bb2]
L1:  max = 10
if i2 < max {
 i3 := i2 + i2;
 goto L1;
}
print(i2);
```

Demo: phi.c

SSA

- Converting from unrestricted form to SSA form

```
1:        i1 = 1
2: L1:    max = 10
3:        if i1 < max {
4:         i2 = i1 + i1
5:         goto L1;
6:        }
7:        print(i);
```

i1 or i2 ?

Phi nodes – special instructions that help deal with control flow. E.g.,
`i3 = phi [i1, bb1], [i2, bb4]`

- Where should we insert phi nodes?

  Dominance frontiers

- What do we need to do after inserting phi nodes?

  Rename variables so that every assignment gets a unique name

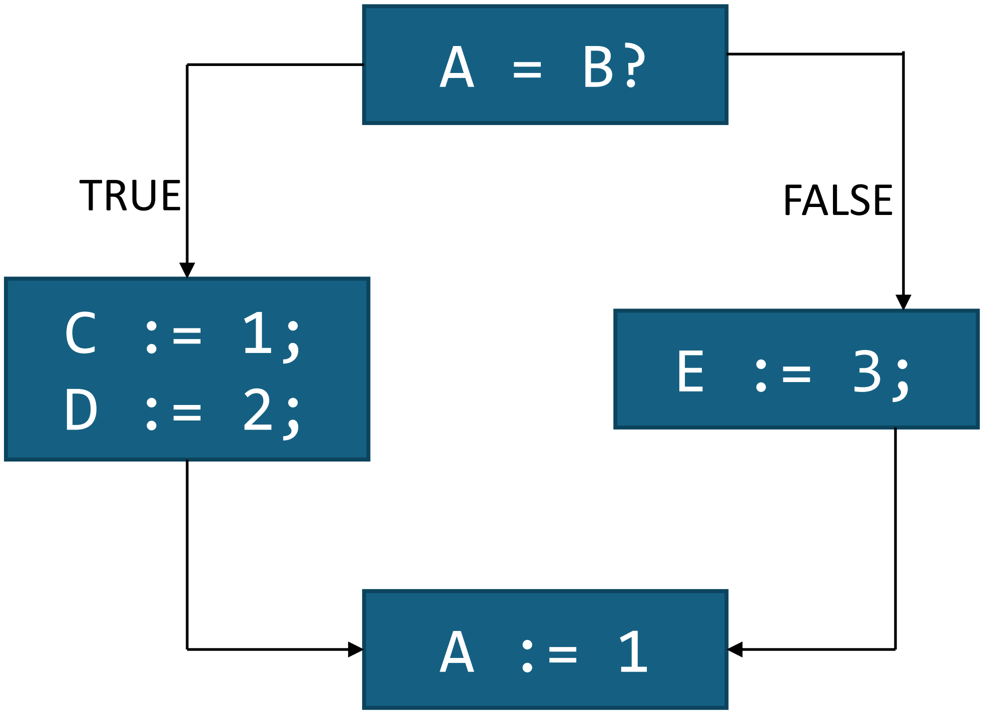
Detour – refresher on basic blocks and CFGs

Basic Blocks and Flow Graphs

- Basic Block
  - Maximal sequence of consecutive instructions with the following properties:
    - The first instruction of the basic block is the *only entry point*
    - The last instruction of the basic block is either the halt instruction or the *only exit point*
- Flow Graph
  - Nodes are the basic blocks
  - Directed edge indicates which block follows which block

Basic Blocks and Flow Graphs - Example

```
if A = B then
   C := 1;
   D := 2;
else
   E := 3
fi
A := 1;
```



A data flow graph

Flow Graphs

- Capture how control transfers between basic blocks due to:
  - Conditional constructs
  - Loops
- Are necessary when we want optimize considering larger parts of the program
  - Multiple procedures
  - Whole program

Flow Graphs - Representation

- We need to label and track statements that are jump targets
  - **Explicit targets** – targets mentioned in jump statement
  - **Implicit targets** – targets that follow conditional jump statement
    - Statement that is executed if the branch is not taken
- Implementation
  - Linked lists for Basic Blocks
  - Graph data structures for flow graphs

End detour

Dominators

- Describe relationship between basic blocks
  - A block dominates other if it is guaranteed to execute before the other
  - **Formally:**
    - if all paths from entry of CFG to node B pass through node A then we say that A dominates B
  - The relationship is reflexive i.e. node B dominates itself

Dominator Tree

- A data structure for tracking dominator relation
- A node in the tree dominates all the nodes of the subtree, for which the node is the root
- Terminology:
  - Strict domination: A strictly dominates B if it dominates B and A ≠ B
  - Immediate domination: A dominates B and does not strictly dominate any other node that strictly dominates B (e.g. A is B's parent in the dominator tree)
  - Domination frontier (DF): B is in the DF of A if
    - A does not dominate B but
    - dominates a predecessor of B
  - Post domination: A post-dominates B if on *all* paths from B to the exit node, A appears.

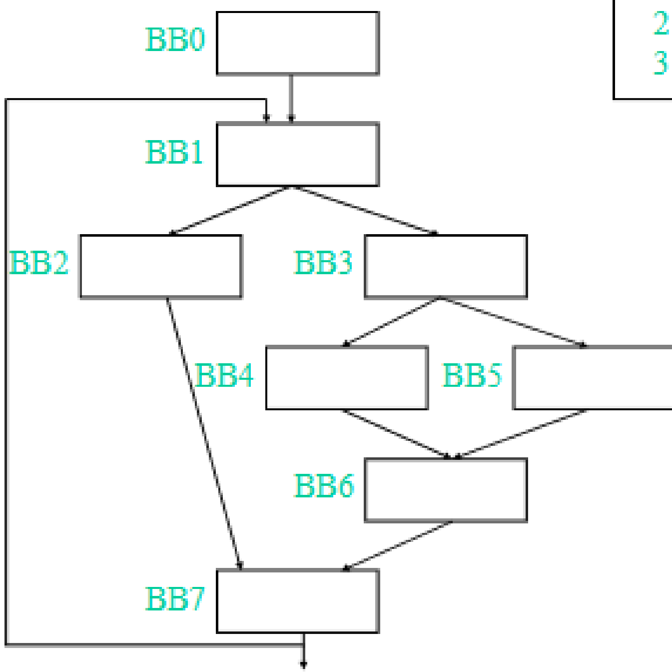
Finding Dominators

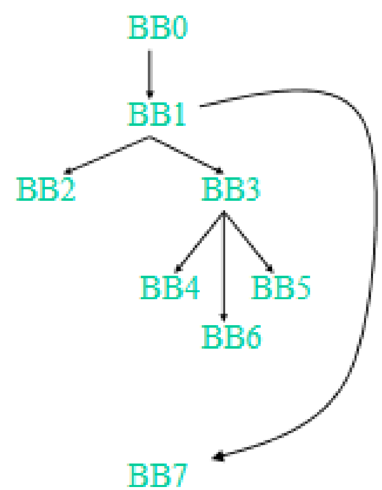
```
Output: dom = {map of vertex -> vertex set}

dom={}
while dom is still changing {
    for vertex in CFG {
```
$$dom[vertex] = vertex \cup \bigcap\{ dom[p], where\ p \in predecessor(vertex)\}$$
```
    }
}
```

Dominator Tree

First BB is the root node, each node dominates all of its descendants

BB	DOM	BB	DOM
0	0	4	0,1,3,4
1	0,1	5	0,1,3,5
2	0,1,2	6	0,1,3,6
3	0,1,3	7	0,1,7





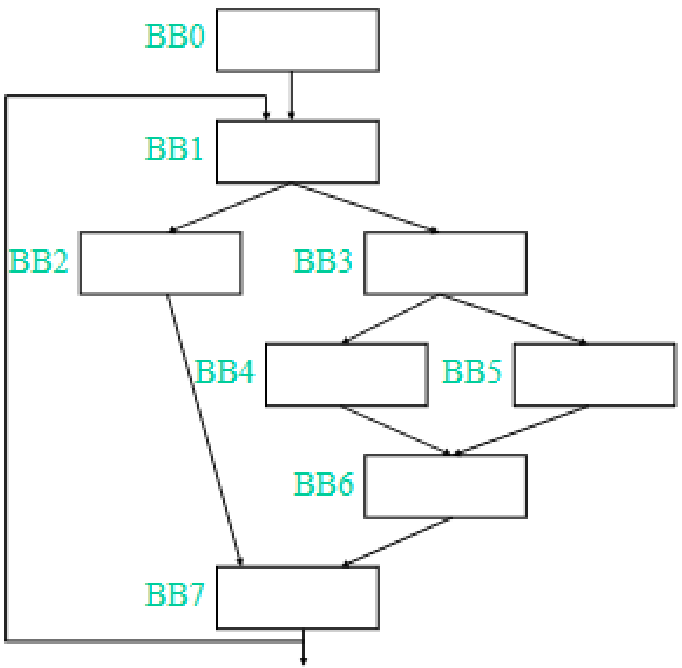
Dom tree

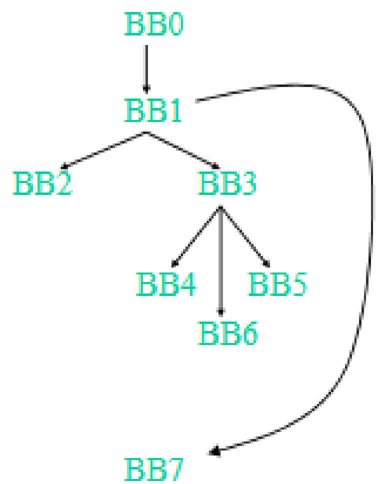
Finding Dominance Frontiers

Recall:

- Dominance frontier of a node X is the set of nodes Y such that
  - X dominates a predecessor of Y
  - X does not strictly dominate Y

Computing Dominance Frontiers





BB	DF
0	-
1	-
2	7
3	7
4	6
5	6
6	7
7	1

For each join point X in the CFG
  For each predecessor of X in the CFG
    Run up to the IDOM(X) in the dominator tree,
    adding X to DF(N) for each N between X and IDOM(X)

Finding Dominators

```
Output: dom = {map of vertex -> vertex set}

dom={}
while dom is still changing {
   for vertex in CFG {
```
$$dom[vertex] = vertex \cup \bigcap\{\, dom[p], where\ p \in predecessor(vertex)\}$$
```
   }
}
```

- Complexity? $O(n^2)$ n is number of vertices.
- Optimization (Linear time):
  - postorder traversal of CFG,
  - reverse the postordered list
  - process vertices in reverse order

  Applicable to programming languages with *reducible* CFGs

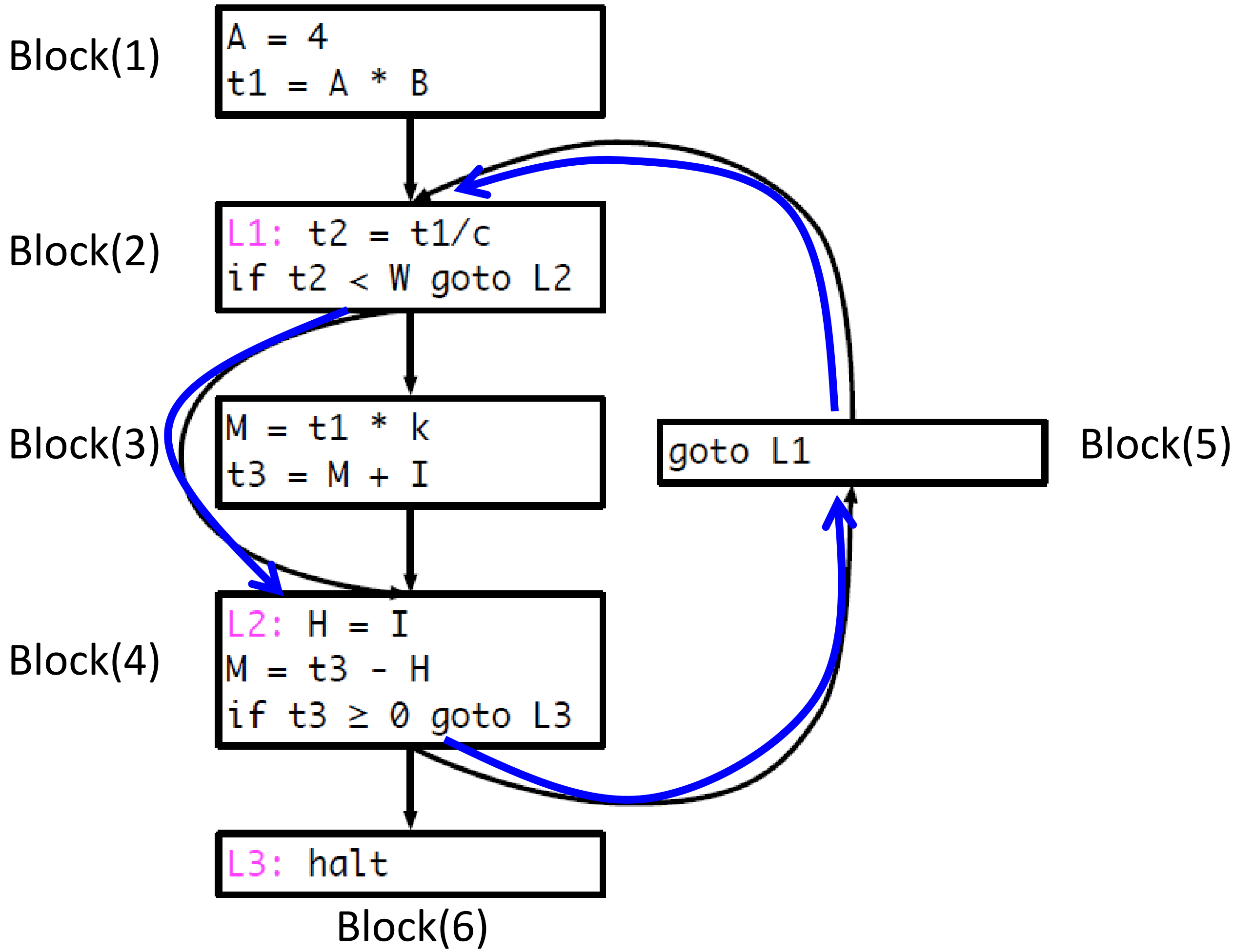
Natural loops

- There is a cycle in the CFG (necessary but not sufficient)
- There is only one vertex with in-edges from outside the set. That one vertex is called the loop-entry/loop-header block. (also the set needs to be strongly connected i.e. there is a path from every vertex to every other vertex in the set).
- For a *backedge A->B,* the *smallest* set of vertices L including A and B such that $\forall\ v \in L,\ predecessors(v) \subseteq L\ or\ v = B$
  - Backedge – an edge from A to B, where B dominates A

  - How to find backedges?

  Do a breadth-first traversal of CFG. Edges that you encounter when discovering new vertices are forward edges. Everything else are backedges.

Exercise: Identify Loops in CFGs



Consider: {B2, B4, B5}. Is this a loop?, Are there other loops?

Reducible CFG

- *If every back edge has a natural loop, then the CFG is reducible.*

- *A programming language with control flow constructs and without goto yields CFGs that are reducible*

Control Flow Graphs

- Recall why do we need CFGs? - Global Optimization
  - Optimizing compilers do global optimization (i.e. optimize beyond basic blocks)
    - Differentiating aspect of normal and optimizing compilers
  - E.g. loops are the most frequent targets of global optimization (because they are often the “hot-spots” during program execution)

Loop Invariant Code Motion

- Optimization that works with natural loops
- Preheader – a unique block that is a predecessor of loop-entry block

Optimize Loops -Identifying Invariant Expressions

- How do we identify expressions that can be moved out of the loop?
  - `LoopDef = {}` set of variables defined (i.e. whose values are overwritten) in the loop body
  - `LoopUse = { }` 'relevant' variables used in computing an expression

```
Mark_Invariants(Loop L) {
   1. Compute LoopDef for L
   2. Mark as invariant all expressions,
      whose relevant variables don't belong
      to LoopDef
}
```

Optimize Loops -Identifying Invariant Expressions

- Example

LoopDef{}

```
for I = 1 to 100          {A, J, K, I}
   for J = 1 to 100       {A, J, K}
      for K = 1 to 100    {A, K}
         A[I][J][K] = (I*J)*K
```

Optimize Loops -Identifying Invariant Expressions

- Example

LoopUse{}

```
for I = 1 to 100              →  {}
   for J = 1 to 100           →  {I}
      for K = 1 to 100        →  {I,J}
         A[I][J][K] = (I*J)*K
```

Optimize Loops -Identifying Invariant Expressions

- Example

Invariant Expressions

```
for I = 1 to 100
   for J = 1 to 100
      for K = 1 to 100
         A[I][J][K] = (I*J)*K
```

{ I*J, Addr(A[i][j])}

For an array access, `A[m] => Addr(A) + m`

For 3D array above*, `Addr(A[I][J][K]) =`

`Addr(A)+(I*10000)-10000+(J*100)-100+K-1`

*Assuming row-major ordering of storage

Optimize Loops -Identifying Invariant Expressions

- Example

Invariant Expressions

```
for I = 1 to 100
   for J = 1 to 100
      for K = 1 to 100
         A[I][J][K] = (I*J)*K
```

{ Addr(A[i]) }

For an array access, `A[m] => Addr(A) + m`

For 3D array above*, `Addr(A[I][J][K]) =`

`Addr(A)+(I*10000)-10000+(J*100)-100+K-1`

*Assuming row-major ordering of storage

Optimize Loops -Factoring Invariant Expressions

- Move the invariant expressions identified

```
Factor_Invariants(Loop L) {
   Mark_Invariants(L);
   foreach expression E marked an invariant:
      1. Create a temporary T
      2. Replace each occurrence of E in L with T
      3. Insert T:=E in L's header code
         // If loop is known to execute at least once,
         insert T:=E before LOOP:
}
```

Optimize Loops -Factoring Invariant Expressions

- Example

```
for I = 1 to 100
   for J = 1 to 100
      for K = 1 to 100
         A[I][J][K] = (I*J)*K        //Invariant Expressions
```

Optimize Loops -Factoring Invariant Expressions

- Example

```
for I = 1 to 100
   for J = 1 to 100
      temp1=A[I][J]
      temp2=I*J
      for K = 1 to 100
         temp1[K] = temp2*K
```

Optimize Loops -Factoring Invariant Expressions

- Example

```
for I = 1 to 100
   temp3=A[I]
   for J = 1 to 100
      temp1=temp3[J]
      temp2=I*J
      for K = 1 to 100
         temp1[K] = temp2*K
```

Optimize Loops -Factoring Invariant Expressions

- Expressions cannot always be moved out!

Case I: We can move `t = a op b` if the statement <u>dominates</u> all loop exits where `t` is live

A node bb1 dominates node bb2 if all paths to bb2 must go through bb1

```
for (...) {
   if(*)
      a = 100
}
c=a
```

Cannot move `a=100` because it does not dominate `c=a` i.e. there is one path (when if condition is false) `c=a` can be executed /'reached' without going to `a=100`

Optimize Loops -Factoring Invariant Expressions

- Expressions cannot always be moved out!

  Case II: We can move `t = a op b` if there is only one definition of `t` in the loop

```
for (...) {
   if(*)
      a = 100
   else
      a = 200
}
```

Multiple definition of a

Optimize Loops -Factoring Invariant Expressions

- Expressions cannot always be moved out!

Case III: We can move `t = a op b` if `t` is not defined before the loop, where the definition reaches t's use after the loop

```
a=5
for (...) {
   a = 4+b
}
c=a
```

Definition of a in a=5 reaches `c=a,` which is defined after the loop

Optimize Loops

- Should be careful while doing optimization of loops

```
while J > I loop
   A(j) := 10/I;
   j := j + 2;
end loop;
```

Optimize Loops – Code Motion

- Should be careful while doing optimization of loops

```
while J > I loop
   A(j) := 10/I;
   j := j + 2;
end loop;
```

- Optimization: can move 10/I out of loop
- What if I = 0?
- What if I != 0 but loop executes zero times?

Optimization Criteria - Safety and Profitability

- Safety - is the code produced after optimization producing same result?
- Profitability - is the code produced after optimization running faster or uses less memory or triggers lesser number of page faults etc.

```
while J > I loop
   A(j) := 10/I;
   j := j + 2;
end loop;
```

- E.g. moving I out of the loop introduces exception (when I=0)
- E.g. if the loop is executed zero times, moving `A(j) := 10/I` out is not profitable

Converting to SSA form

- Where do we insert phi nodes?

```
for v in vars {
    for d in defs[v] {
        for block in DF[d] {
            if (block does not have a phi node)
                add phi node to block
            if (block is not part of defs[v])
                add block to defs[v]
        }
    }
}
```

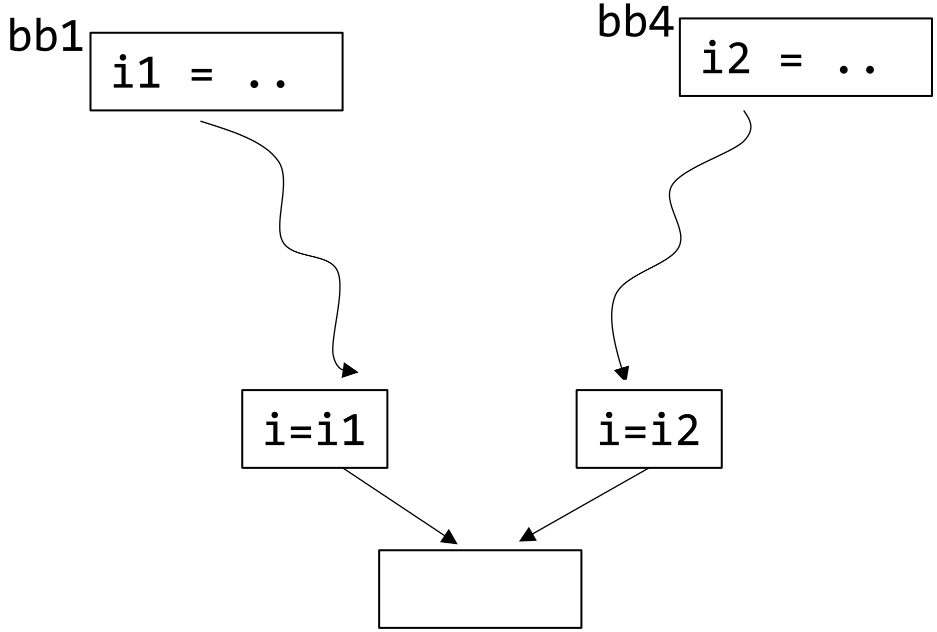
Converting to SSA form

- Rename variables

```
Stack[v] is a stack of variable names for every variable v

Rename(block) {
    foreach instn in block {
        replace each argument to instn with stack[argument's oldname]
        replace instn's destination with newname
        push newname to stack[destination's oldname]
    }

   foreach s in successor(block){
        foreach p in s's phi nodes {
            read stack[v] and plug in p (if p is for v)
        }
   }

   foreach b in immediately_dominated_by(block)//need dominance tree here
        Rename(b)

   pop all names pushed to stack
}

Rename(entry)
```

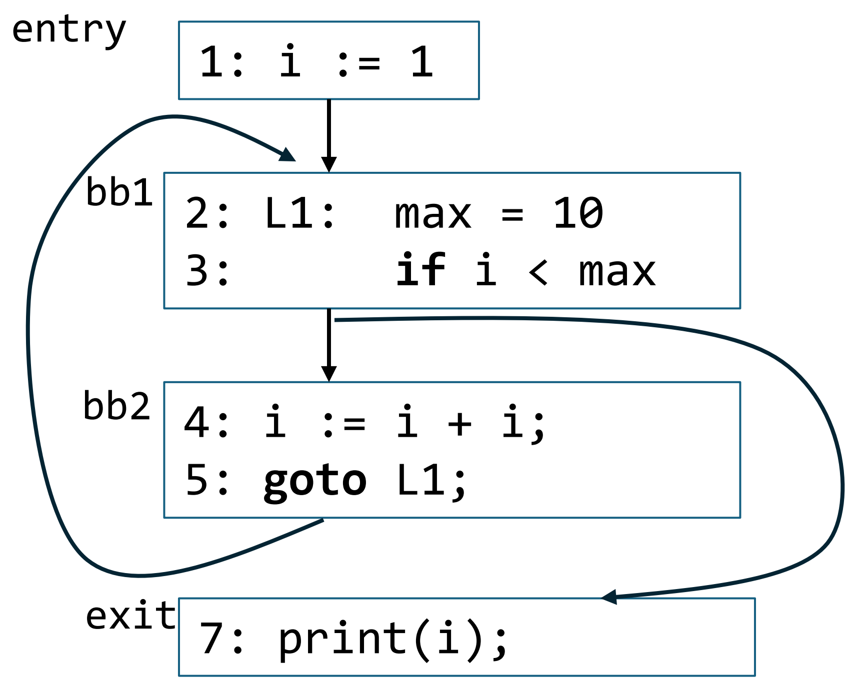
Converting **from** SSA form

- Get rid of Phi nodes



Acknowledgements

- Compiler Explorer (browser-based option to view LLVM IR with phi nodes. Use `-O1 -emit-llvm` compiler flag with `phi.c`)
- https://www.llvmpy.org/llvmpy-doc/dev/doc/llvm_concepts.html
- CS 6120: The Self-Guided Course (lesson 5)


entry
1: i := 1
bb1
2: L1: max = 10
3: if i < max
bb2
4: i := i + i;
5: goto L1;
exit
7: print(i);