ECE 6775
High-Level Digital Design Automation
Fall 2023

More CFG
Static Single Assignment

Cornell University

CSL

Announcements

- Lab 2 is released

Agenda

- More control flow analysis
  - Dominator tree
  - Dominance frontier

- Dataflow analysis – static single assignment (SSA)
  - SSA definition
  - PHI node (Φ-node) placement
  - Code optimizations with SSA

- A brief overview of LLVM

Review: Dominance Relation

- **Definition:** Let G = (V, E, s) denote a CFG, where

  V : set of nodes

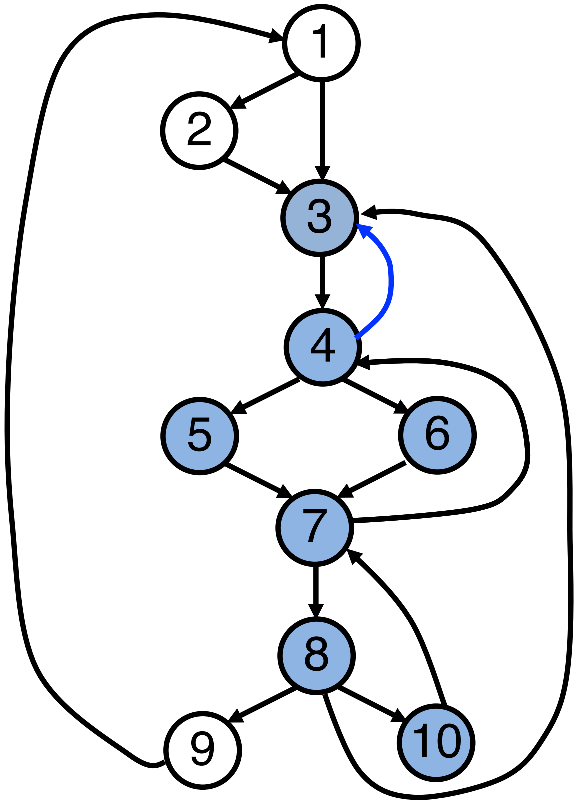
  E : set of edges

  s : entry node and

  let $p \in V$, $q \in V$

  - p **dominates** q, written $p \leq q$
    - also written $p \in DOM(q)$
  - p **properly (strictly) dominates** q, written $p < q$ if $p \leq q$ and $p \neq q$
  - p **immediately** (or directly) **dominates** q, written $p <_d q$
    if $p < q$ and there is no $t \in V$ such that $p < t < q$
    - also written $p = IDOM(q)$

Review: Finding Loops



Find all back edges in this graph and the natural loop associated with each back edge

(9,1)	Entire graph
(10,7)	{7,8,10}
(7,4)	{4,5,6,7,8,10}
(3,4)	{3,4,5,6,7,8,10}

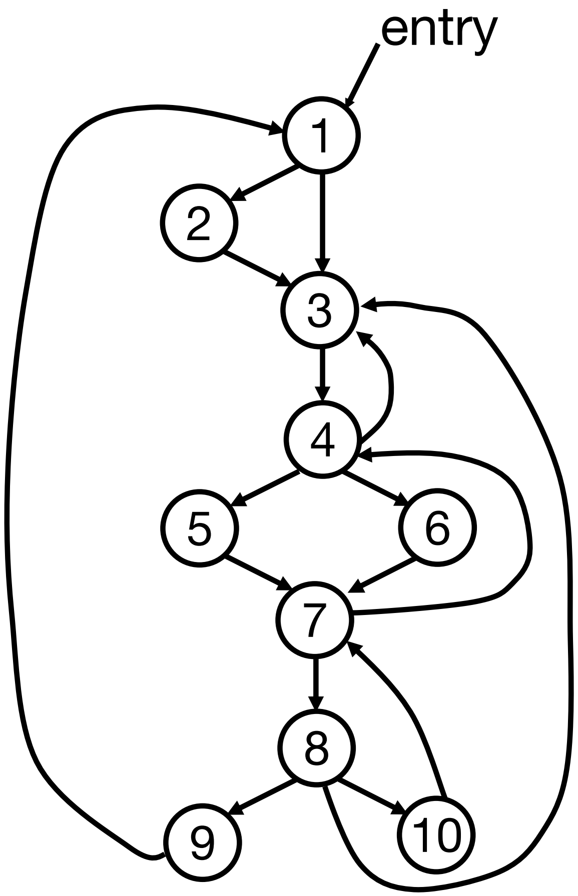
Dominator Tree

- A node (basic block) N in CFG may have multiple dominators, but **only one of them will be closest to N** and be dominated by all other dominators of N

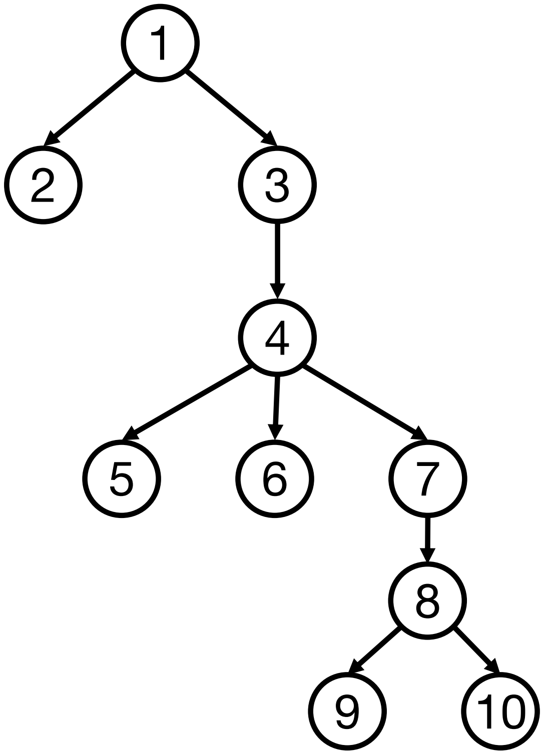
- A dominator tree is a useful way to represent the dominance relation
  - The entry node *s* is the root
  - **Each node in the dominator tree is the immediate dominator of its children**
    - Each node *d* dominates only its descendants in the tree

Example: Dominator Tree

CFG



Dominator Tree



Dominance Frontier

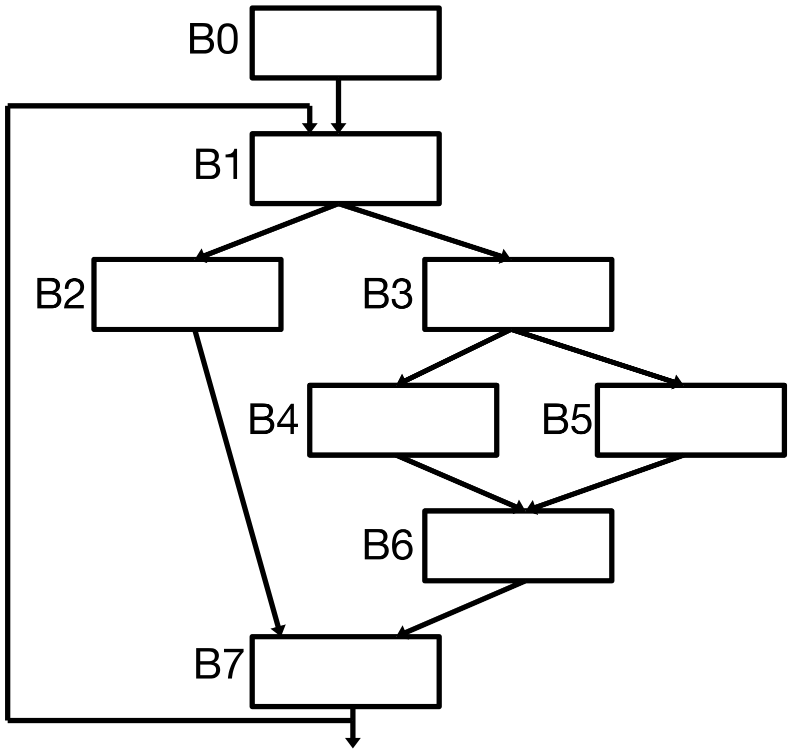
- A basic block q is in the **dominance frontier set (DF)** of basic block p if and only if
  - (1) p *does NOT strictly dominate* q
  - (2) p *dominates some predecessor(s)* of q
  - If above two conditions hold, $q \in DF(p)$

- Intuitively, for an $q \in DF(p)$, q is <u>almost strictly dominated</u> by p

- Useful for efficiently computing the SSA form

Example: Dominance Frontiers

CFG



Dominance frontiers (DF)

p	**DF(p)**
0	**–**
1	**?**
2	**7**
3	**7**
4	**6**
5	**6**
6	**7**
7	**?**

q is in the dominance frontier set (DF) of p iff

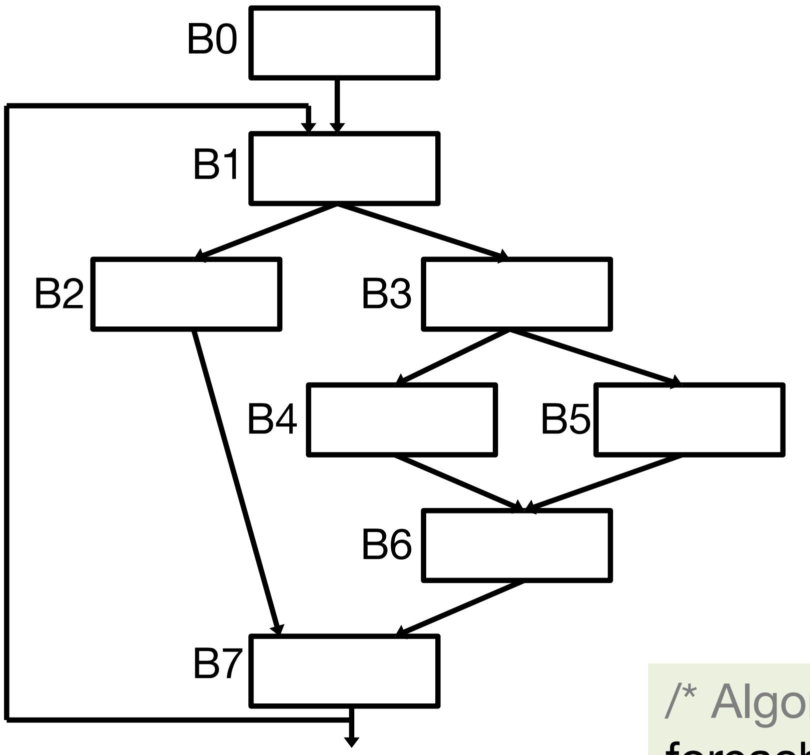
(1) p does NOT strictly dominate q

(2) p dominates some predecessor(s) of q

If above two conditions hold, $q \in DF(p)$

Dominance Frontiers and Dominator Tree

CFG

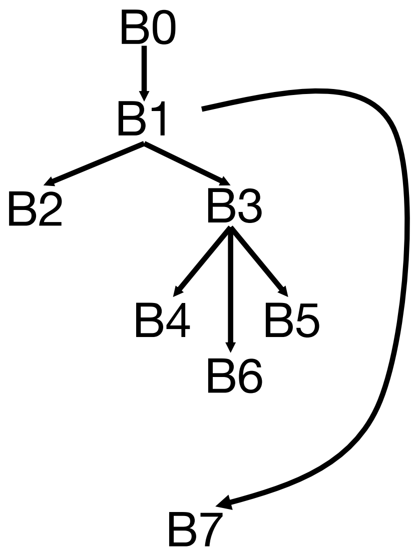


[1] **convergence point** is a node in CFG with multiple predecessors

Dominance frontiers (DF)

p	**DF(p)**
0	**–**
1	**1**
2	**7**
3	**7**
4	**6**
5	**6**
6	**7**
7	**1**

Dominator tree



```
/* Algorithm to construct the DF sets */
foreach convergence point¹ Q in CFG
    foreach predecessor X of Q in CFG
        Run up to Y=IDOM(Q) in the dominator tree,
        adding Q to DF(P) for each P between [X, Y)
```

Only convergence points are added to the DF sets!

Static Single Assignment

- Static single assignment (**SSA**) form is a restricted IR where
  - Each variable <u>def</u>inition has a **unique** name
  - Each variable <u>use</u> refers to a single definition

- SSA simplifies **data flow analysis** and many compiler optimizations
  - Eliminates artificial dependences (on scalars)
    - Write-after-write
    - Write-after-read

SSA within a Basic Block

- **Assign each variable definition a unique name**
- Update the uses accordingly

Original code

defs of x

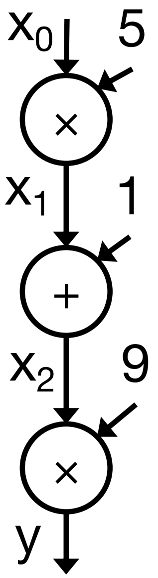
```
x = read()
x = x * 5
x = x + 1
y = x * 9
```

uses of x

SSA form

x_0 = read()
$x_1 = x_0 * 5$
$x_2 = x_1 + 1$
$y = x_2 * 9$

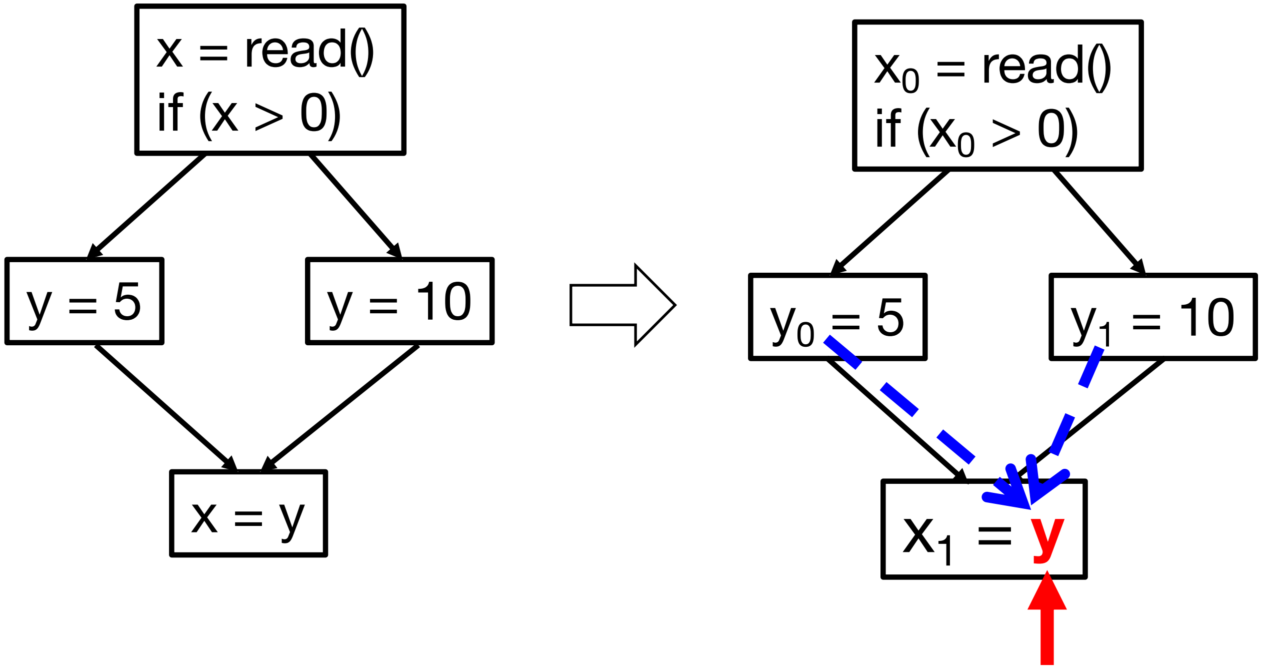
Corresponding data flow graph



SSA with Control Flow

- Consider a situation where two control-flow paths merge
  - e.g., due to an if-then-else statement or a loop

```
x = read()
if (x > 0)
    y = 5
else
    y = 10
x = y
```



x = read()
if (x > 0)

y = 5

y = 10

x = y

x_0 = read()
if (x_0 > 0)

y_0 = 5

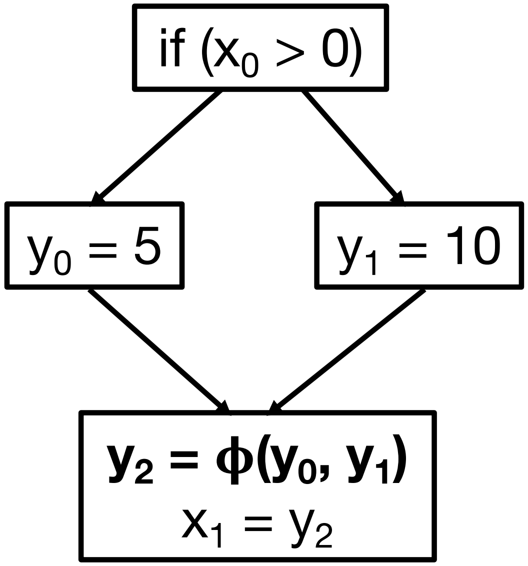
y_1 = 10

x_1 = y

should this be y_0 or y_1?

Introducing ϕ-Node

- Inserts special join functions (called **ϕ-nodes** or PHI nodes) at points where different control flow paths converge



if ($x_0 > 0$)

$y_0 = 5$

$y_1 = 10$

$\mathbf{y_2 = \phi(y_0, y_1)}$

$x_1 = y_2$

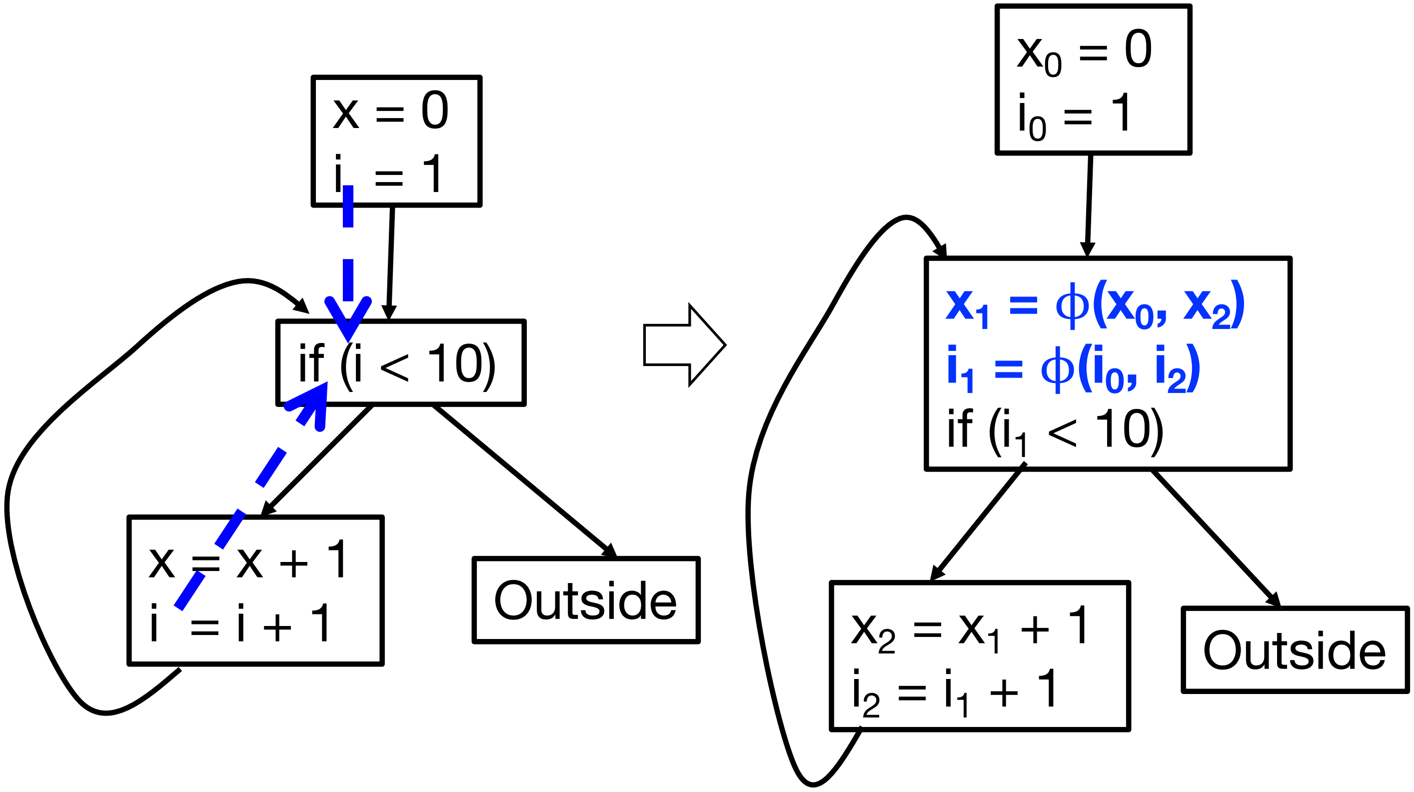
Note: ϕ is not an executable function!

To generate executable code from this form, appropriate copy statements need to be generated in the predecessors (in other words, reversing the SSA process for code generation)

SSA in a Loop

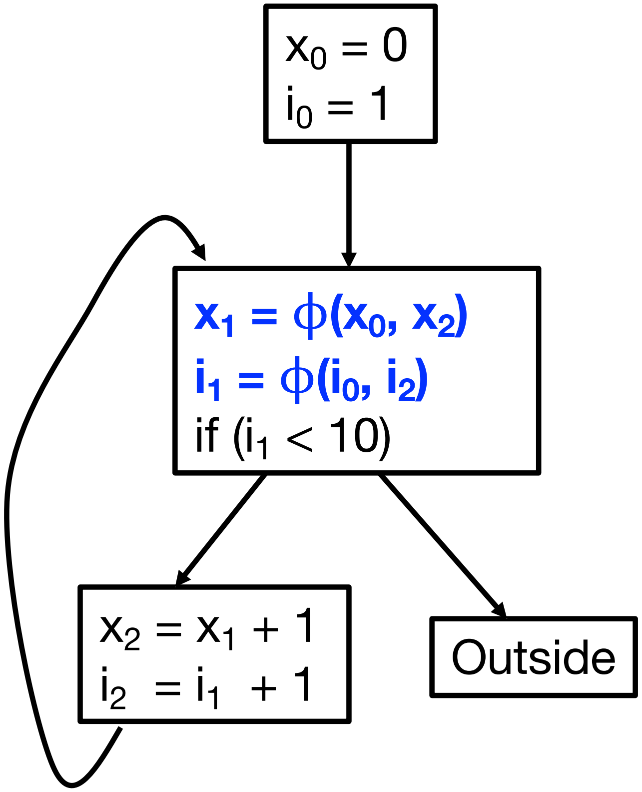
- Insert ϕ-nodes in the loop header block

```
x = 0
i = 1
while (i<10) {
     x = x+i
     i = i+1
}
```



ϕ−Node Placement

- When and where to insert ϕ-nodes?
  - If two control paths A → C and B → C converge at a node C, and both A and B contain assignments to variable “x”, then ϕ-node for “x” must be placed at C
    - **We call C a join node or convergence point**
    - Generalizes to more than two converging control paths
- Objective: Minimize the number of ϕ-nodes
  - Need to compute dominance frontier sets

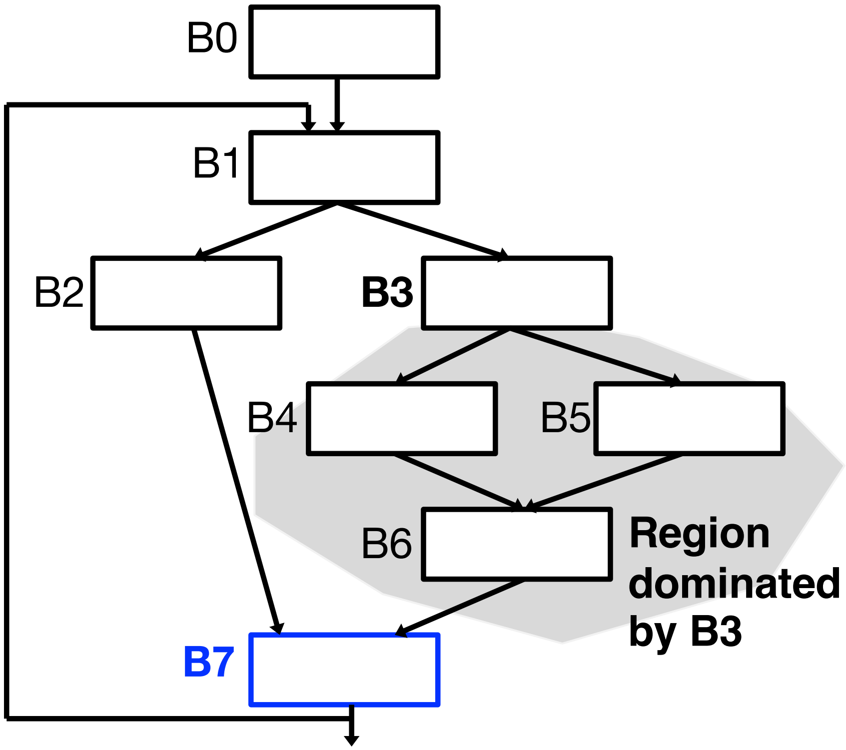


$x_0 = 0$
$i_0 = 1$

$x_1 = \phi(x_0, x_2)$
$i_1 = \phi(i_0, i_2)$
if ($i_1 < 10$)

$x_2 = x_1 + 1$
$i_2 = i_1 + 1$

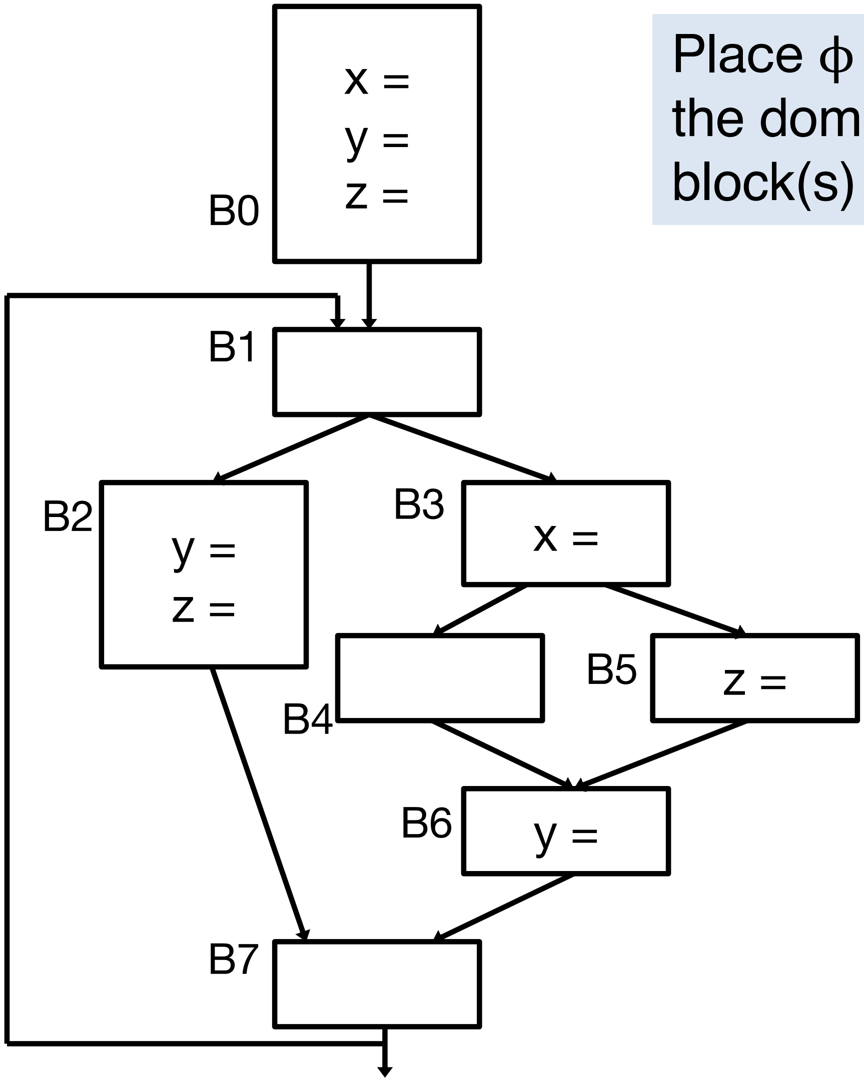
Example: Dominance Frontier and SSA



- B7 is in the dominance frontier set of B3
  - In other words, B7 is the destination of some edge(s) leaving a region dominated by B3
- **For each variable definition in B3, a ϕ node is needed in B7**

ϕ-Node Placement

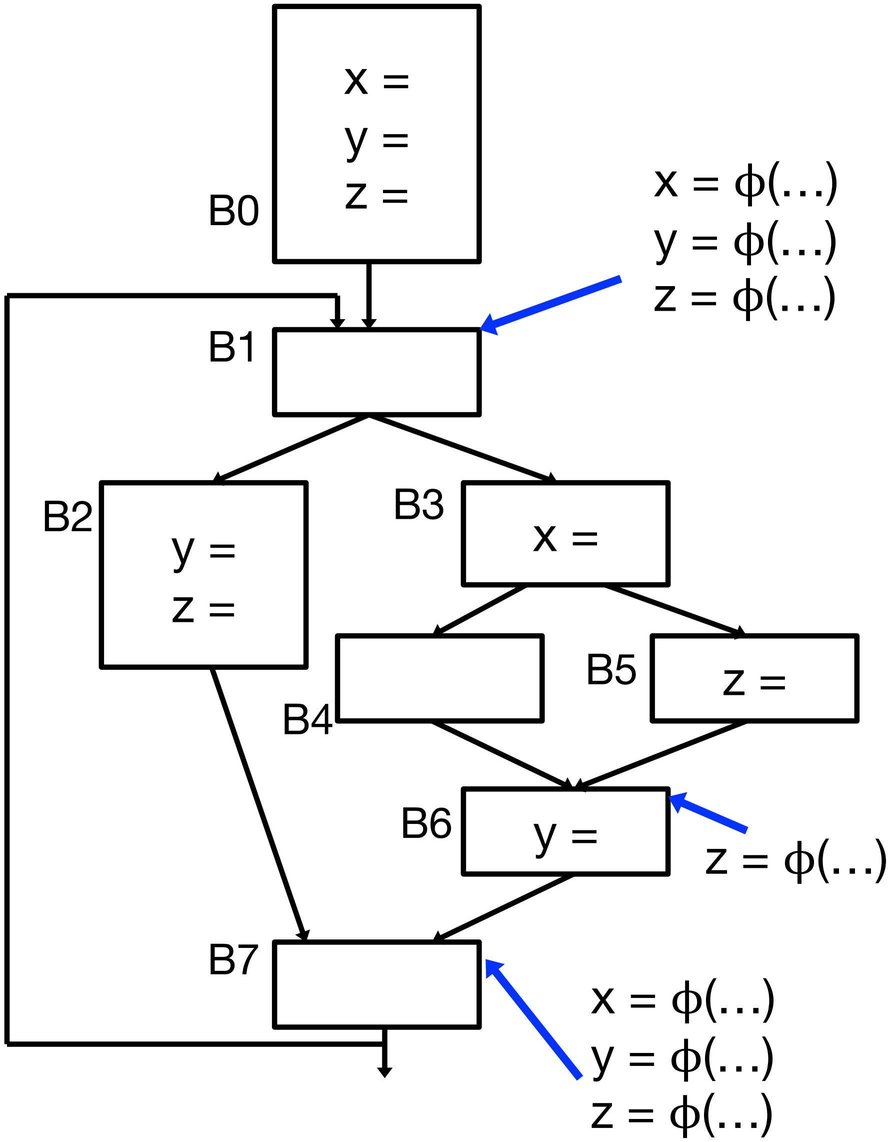
p	**DF**
0	-
1	1
2	7
3	7
4	6
5	6
6	7
7	1



Place ϕ node(s) of a variable **x** in the dominance frontier set of the block(s) where **x** gets defined

ϕ-Node Placement: Iterative Insertion

p	**DF**
0	-
1	1
2	7
3	7
4	6
5	6
6	7
7	1



- **x** is defined in 0, 3
  => insert ϕ in 7,
  then **x** also defined in 7
  => insert ϕ in 1
- **y** is defined in 0, 2, 6
  => insert ϕ in 7
  then **y** also defined in 7
  => insert ϕ in 1
- **z** is defined in 0,2,5
  => insert ϕ in 6,7
  then **z** also defined in 7
  => insert ϕ in 1

Afterwards, assign a unique name to each variable definition (including ϕ nodes) and update all uses

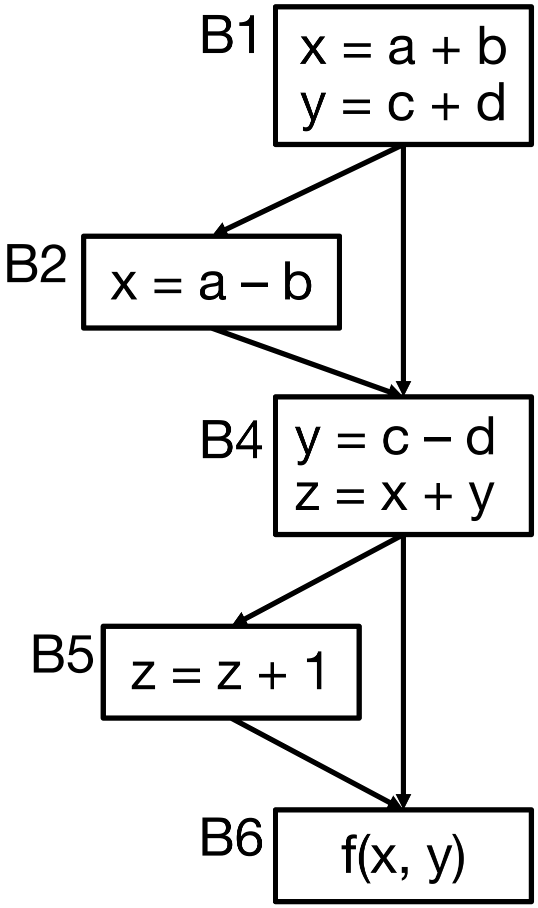
SSA Applications

- SSA form simplifies data flow analysis and many code transformations
  - Primarily due to explicit & simplified (sparse) def-use chains

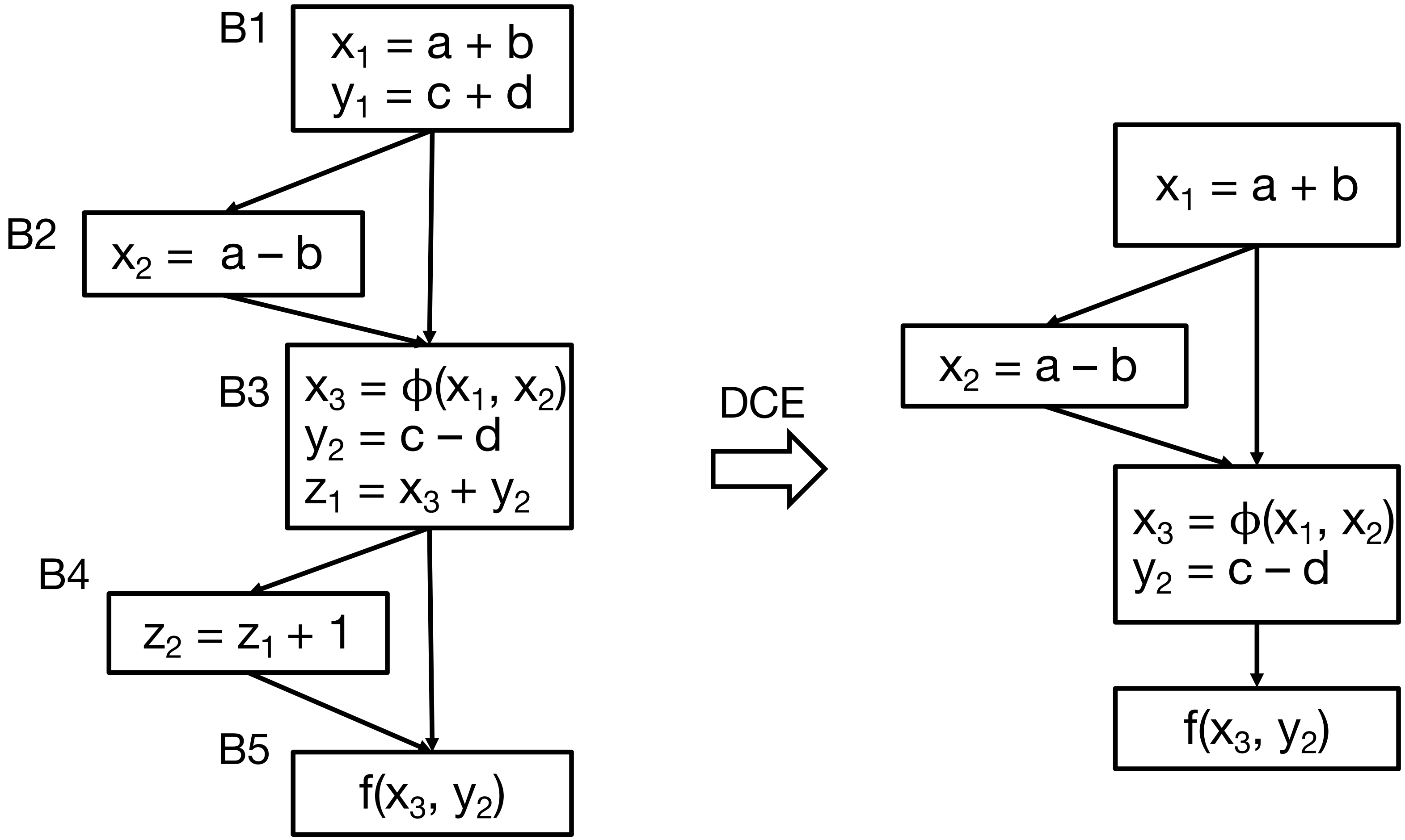
- Here we show two simple examples
  - Dead code elimination
  - Loop induction variable detection

Dead Code in CDFG

- A dead statement is either
  - (1) Unreachable code
  - (2) Definitions never used
- How to efficiently Identify the dead statements?



Dead Code Elimination (DCE) with SSA



B1

$x_1 = a + b$

$y_1 = c + d$

B2

$x_2 = a - b$

B3

$x_3 = \phi(x_1, x_2)$

$y_2 = c - d$

$z_1 = x_3 + y_2$

B4

$z_2 = z_1 + 1$

B5

$f(x_3, y_2)$

DCE

$x_1 = a + b$

$x_2 = a - b$

$x_3 = \phi(x_1, x_2)$

$y_2 = c - d$

$f(x_3, y_2)$

Iteratively remove unused definitions

first remove y1, z2, and B4; then remove z1

Loop Induction Variables

- An induction variable is a variable that
  - Gets increased or decreased by a fixed amount (loop invariant) on every iteration of a loop
    - i = i + c (**basic induction variable**)
  - or is an affine function of another induction variable
    - j = a * i + b (**mutual induction variable**)

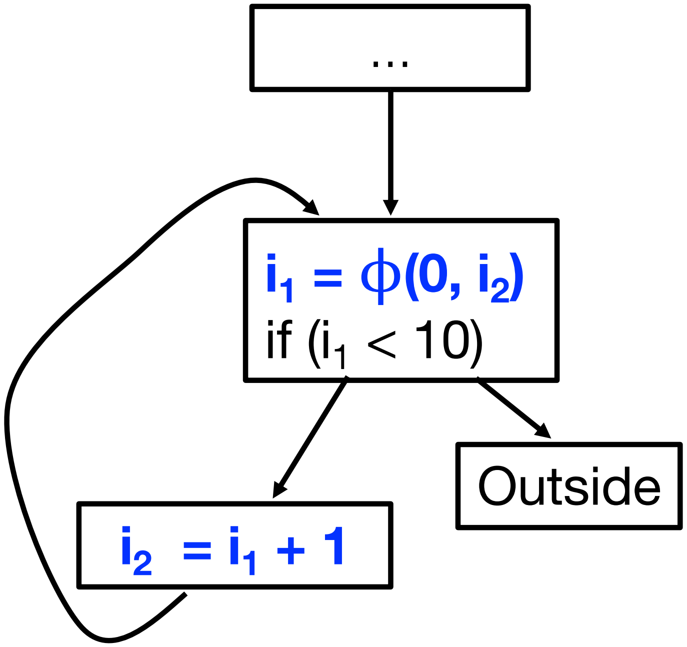
Identifying Basic Loop Induction Variable

- Find basic loop induction variable(s)

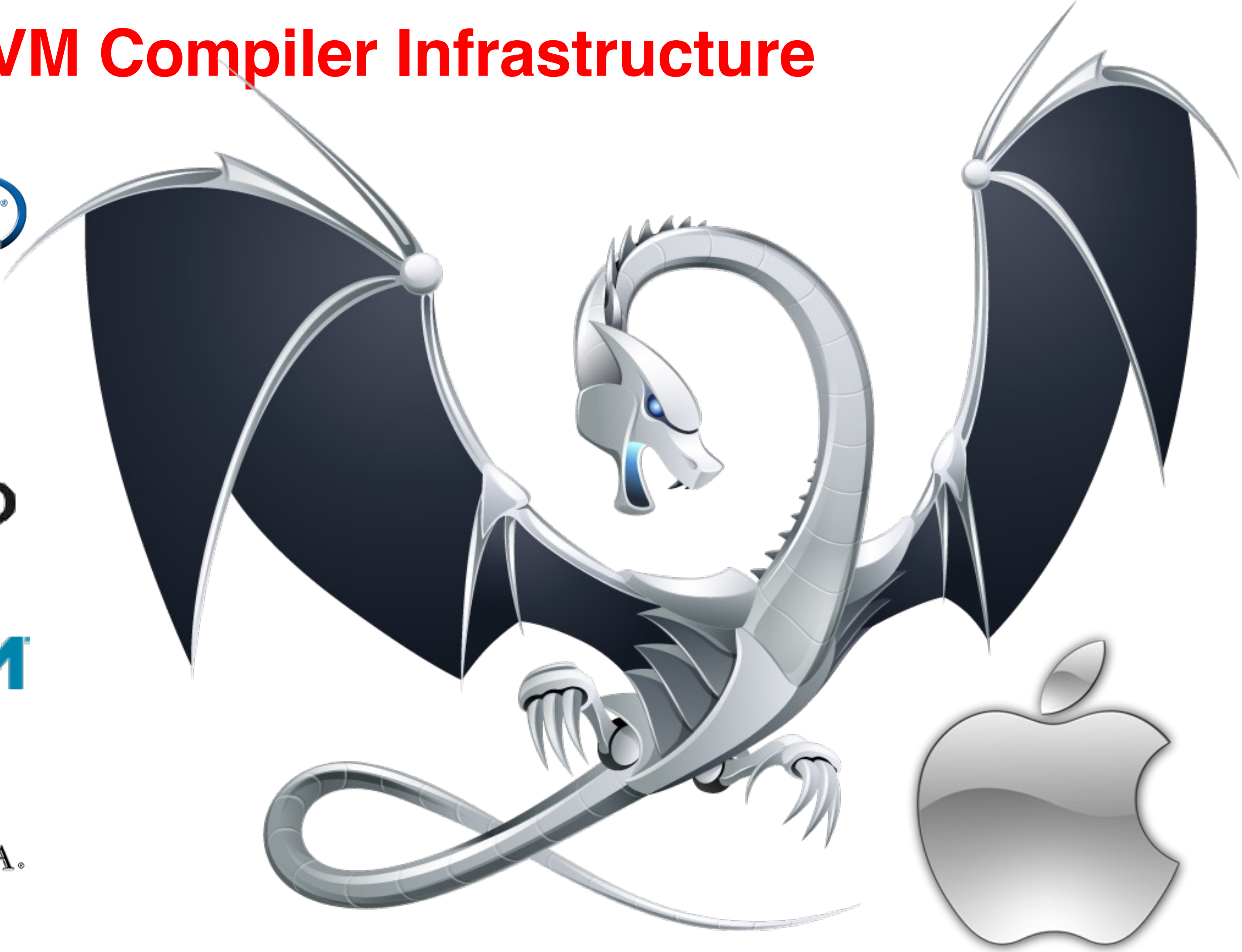
  1. Inspect back edges in the loop

  2. Each back edge points to a ϕ node in the loop header, which may indicate a basic induction variable

  3. ϕ is a function of an initialized variable and a definition in the form of "i + c" (i.e., increment operation)



LLVM Compiler Infrastructure



intel

AMD

ARM

NVIDIA

Qualcomm snapdragon

EA

Adobe

LLVM is not Compiler but a Compiler Infrastructure

What is LLVM?

- Formerly Low Level Virtual Machine
  - Brainchild of Chris Lattner and Vikram Adve back in 2000
  - ACM Software System Award in 2012

- **The core of LLVM is the SSA-base IR**
  - Language independent, target independent, easy to use
  - RISC-like virtual instructions, unlimited registers, exception handling, etc.

- Provides modular & reusable components for building compilers
  - Components are ideally language/target independent
  - Allows choice of the right component for the job
  - Many high-quality libraries (components) with clean interfaces
    - Optimizations, analyses, modular code generator, profiling, link time optimization, ARM/X86/PPC/SPARC code generator …
    - Tools built from the libraries: C/C++/ObjC compiler, modular optimizer, linker, debugger, LLVM JIT …

The Structure of a Program in LLVM

- Module contains Functions/GlobalVariables
  - Module is unit of compilation/analysis/optimization
- Function contains BasicBlocks/Arguments
  - Functions roughly correspond to functions in C
- BasicBlock contains list of instructions
  - Each block ends in a control flow instruction
- Instruction is opcode + vector of operands
  - All operands have types
  - Instruction result is typed

Example: An LLVM Loop

```
for (i=0; i<N; ++i)
  foo(A[i], &P);
```

```
loop:
  %i.1 = phi i5 [ 0, %bb0 ], [ %i.2, %loop ]
  %AiAddr = getelementptr float* %A, i32 %i.1
  call void %foo(float %AiAddr, %pair* %P)
  %i.2 = add i5 %i.1, 1
  %tmp = icmp eq i5 %i.1, 16
  br i1 %tmp, label %loop, label %outloop
```

- High-level information exposed in the code
  - Explicit dataflow through SSA form
  - Explicit control-flow graph
  - Explicit language-independent type-information
  - Explicit typed pointer arithmetic
    - Preserve array subscript and structure indexing

source: http://llvm.org

Arbitrary Precision Integers in LLVM

- LLVM is adopted in several commercial and academic HLS tools
- It has built-in support for arbitrary width integers since version 2.0 (e.g., i2, i128, i1024)
  - Essential for hardware synthesis
  - An 11b multiplier is significantly cheaper/faster than a 16b implementation
  - Can leverage other LLVM analyses/optimizations to perform bitwidth minimization

LLVM Flow Analysis

- LLVM IR is in SSA form
  - use-def and def-use chains are always available
  - All objects have user/use info, even functions
- Control flow graph (CFG) is always available
  - Exposed as BasicBlock predecessor/successor lists
  - Many generic graph algorithms usable with the CFG
- Higher-level info implemented as passes
  - CallGraph, Dominators, LoopInfo, …

source: http://llvm.org

Next Lecture

- Scheduling

Acknowledgements

- These slides contain/adapt materials developed by
  - Prof. Scott Mahlke (UMich)
  - Dr. Chris Lattner (Modular AI)