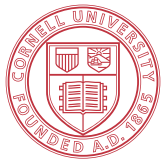
ECE 5775
High-Level Digital Design Automation
Fall 2018

Control Flow Graph



Cornell University

CSL

Announcements

- First problem set (HW1) is posted
  - Due Monday 9/17

- Lab 2 will be released on Thursday

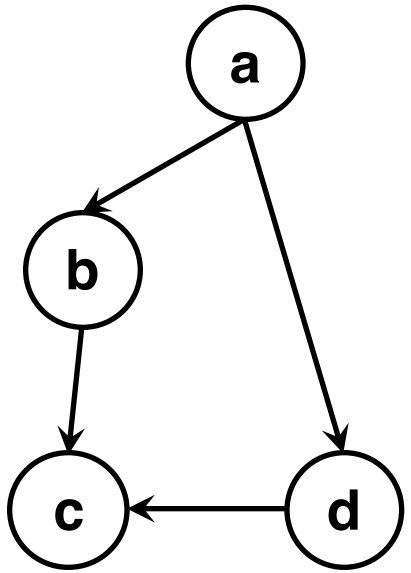
Revisiting Some Important Graph Definitions

- What is a strongly connected component (SCC)?
- Is a DAG strongly connected? No
- Does topological search apply to an SCC? No
- What is the time complexity of a topological search (in terms of |V| and |E|)? O(|V|+|E|)

Graph Traversal

- Purpose: visit all the vertices in a particular order, check/update their properties along the way

- Algorithms
  - Depth-first search (DFS)
  - Breadth-first search (BFS)
  - Either can be used to realize topological sort



a→b→c→d

DFS order = a → ?

a→b→d→c

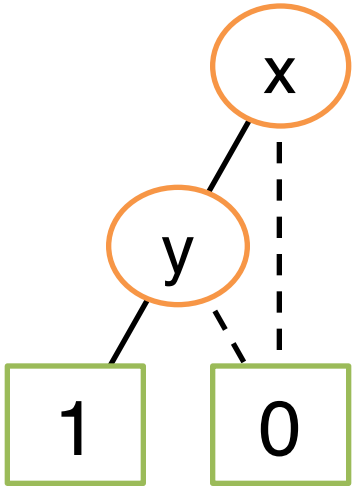
BFS order = a → ?

Outline

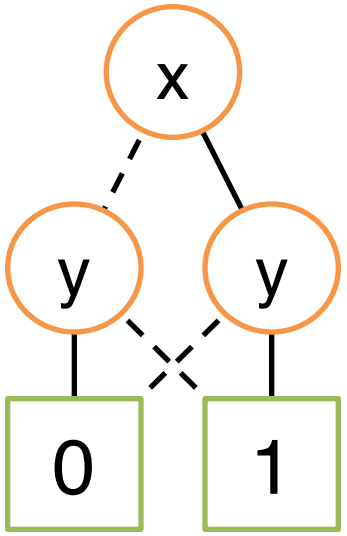
- More on BDDs and static timing analysis
- Basics of control data flow graph
  - Basic blocks
  - Control flow graph
- Dominance relation
  - Finding loops

Review: Two-Input Gates in BDD

Name the logic gates represented by the following BDDs



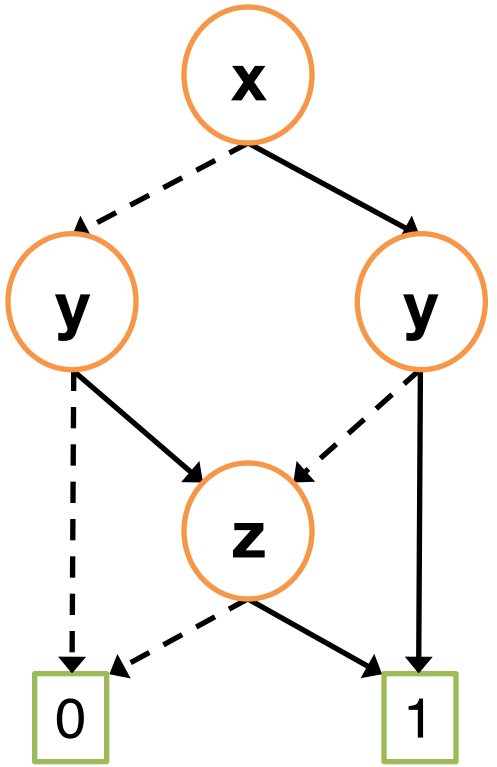
(a) AND



(b) XNOR

More Virtues of BDDs

- There are many, but to list a few more:
  - Can represent an exponential number of paths with a DAG
  - Can evaluate an *n*-ary Boolean function in at most *n* steps
    - By tracing paths to the 1 node, we can count or enumerate all solutions to equation f = 1
  - Every BDD node (not just root) represent some Boolean function in a canonical way
    - A BDD can be multi-rooted representing multiple Boolean functions sharing subgraphs



BDD Application on Functional Verification

```
bool P(bool x, bool y) { return ~(~x & ~y); }
```

```
bool Q(bool x, bool y) { return x ^ y; }
```

- Either prove equivalence
- Or find counterexample(s)
  - Input values (x, y) for which these programs produce different results

Straight-line evaluation

```
v1 = ~x
v2 = ~y
v3 = v1 & v2
P = ~v3
```

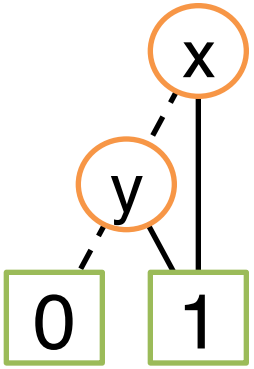
```
Q = x ^ y
```

Is (P == Q) true?

BDD-based Equivalence Checking

P = ~v3 = ~(~x & ~y) = x | y

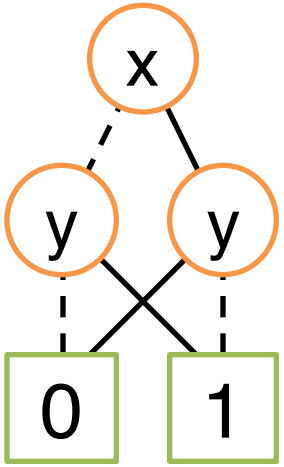
OR gate



BDD of P

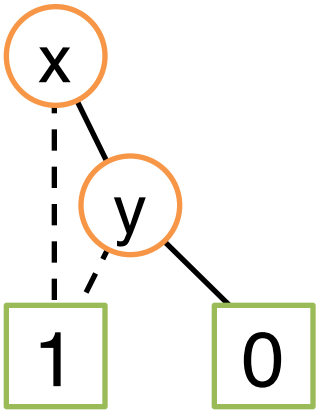
Q = x ^ y

XOR gate



BDD of Q

T = (P == Q)



BDD of T

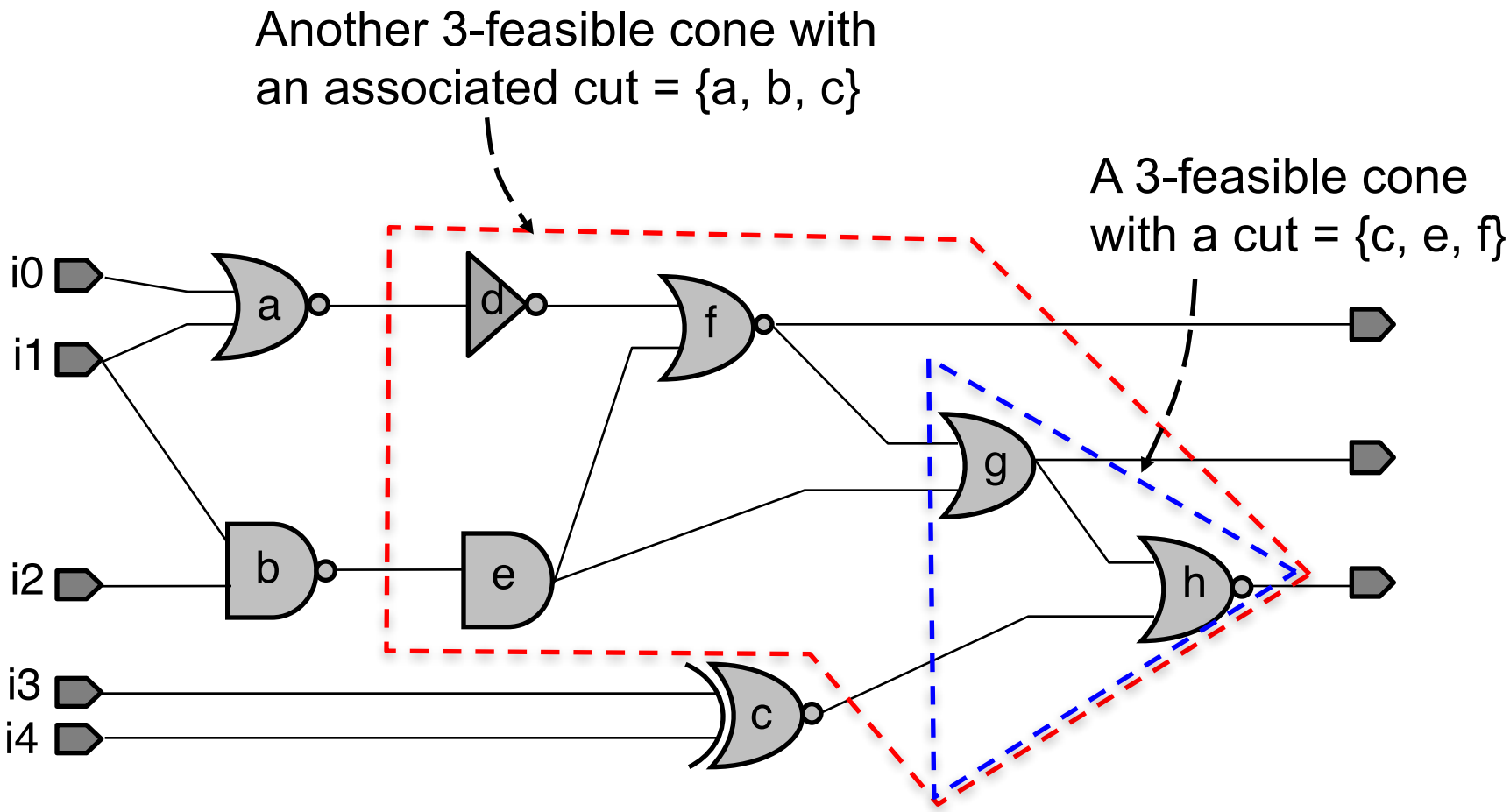
Counterexample

Setting x = 1 & y = 1 leads to false output

Hence P does not equal Q

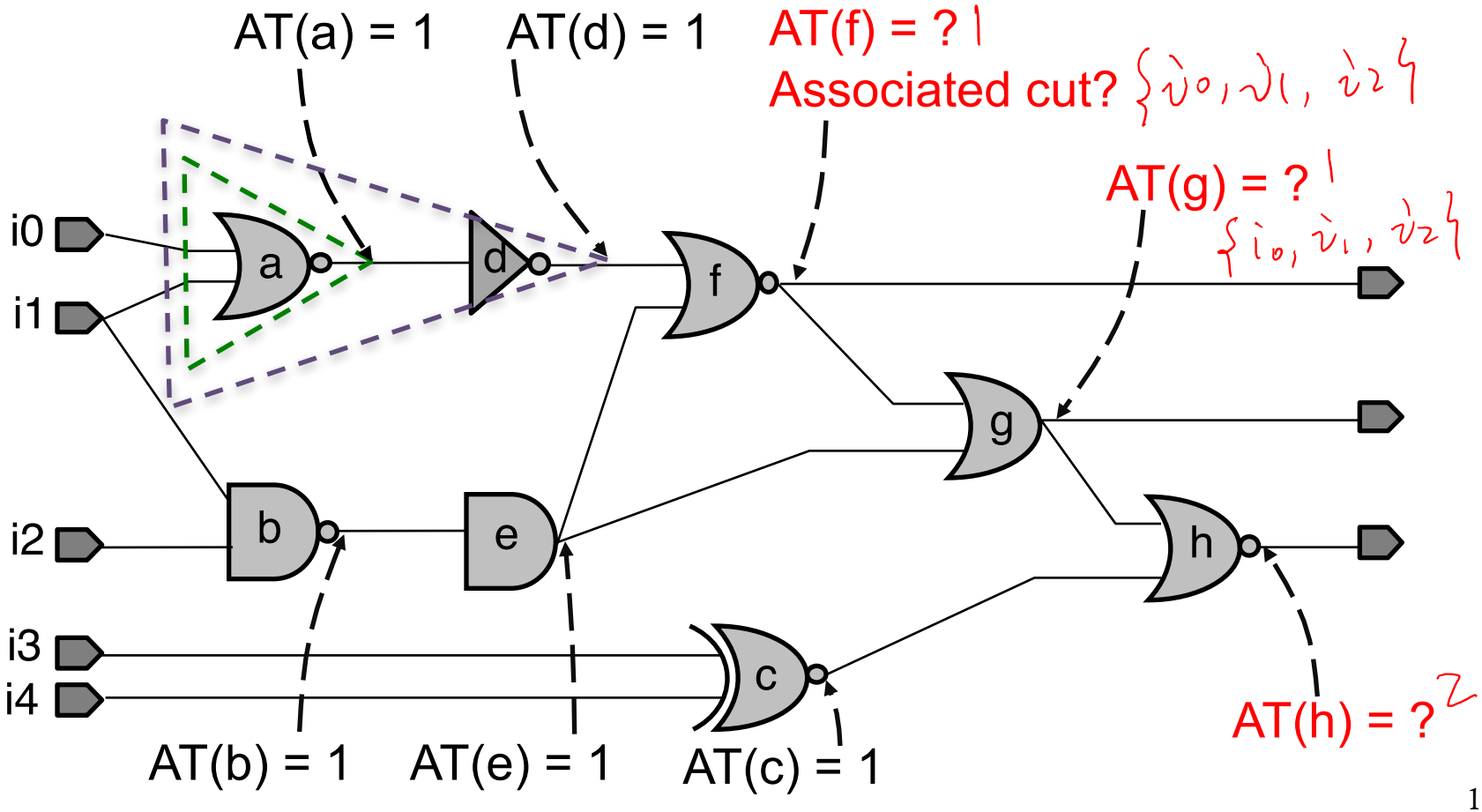
FPGA LUT Mapping Revisited

- Cone C_v : a subgraph rooted on a node v
  - K-feasible cone: #inputs(C_v) ≤ K (**Can occupy a K-input LUT**)
  - K-feasible cut: The set of input nodes of a K-feasible C_v

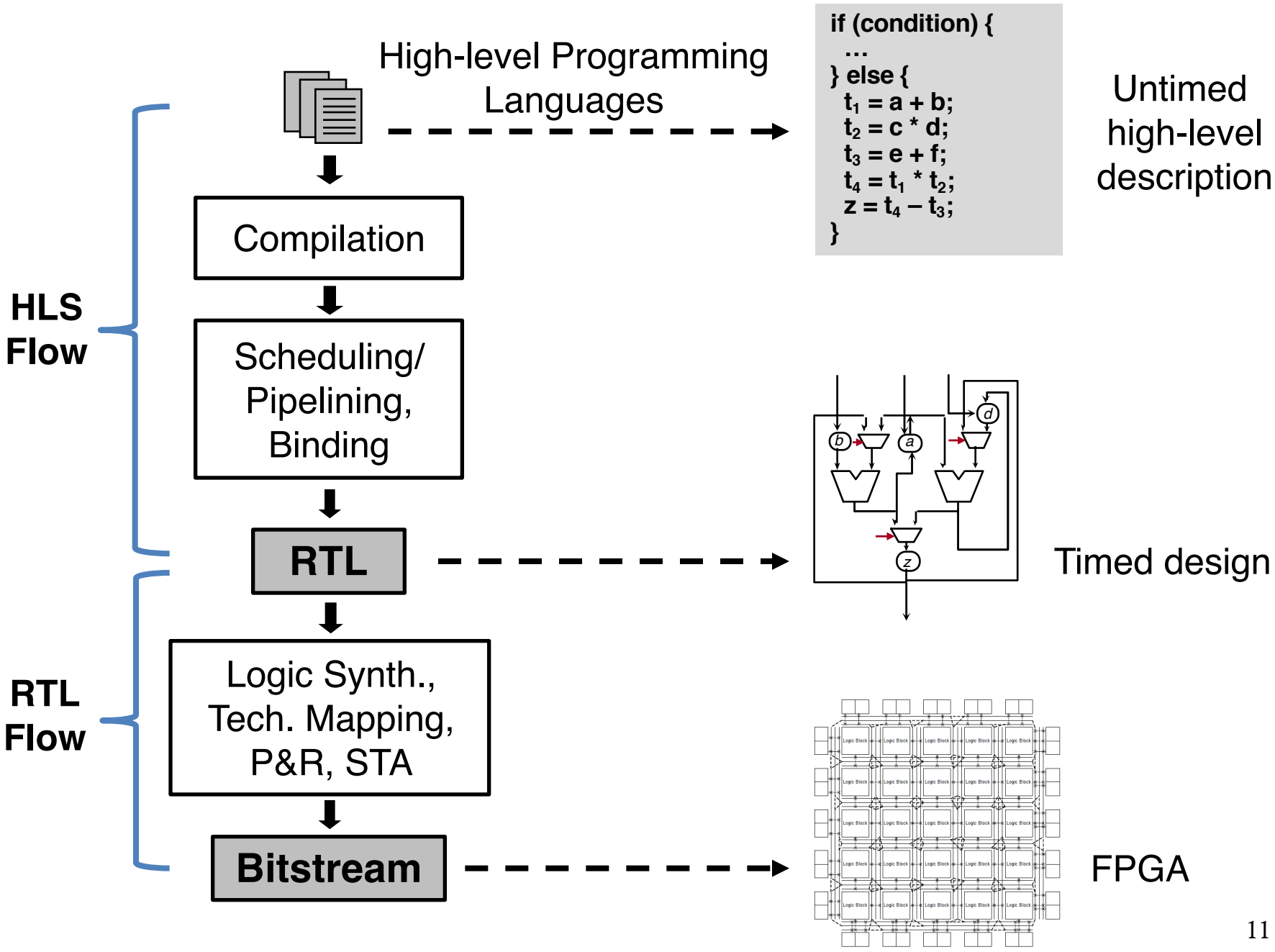


Timing Analysis with LUT Mapping

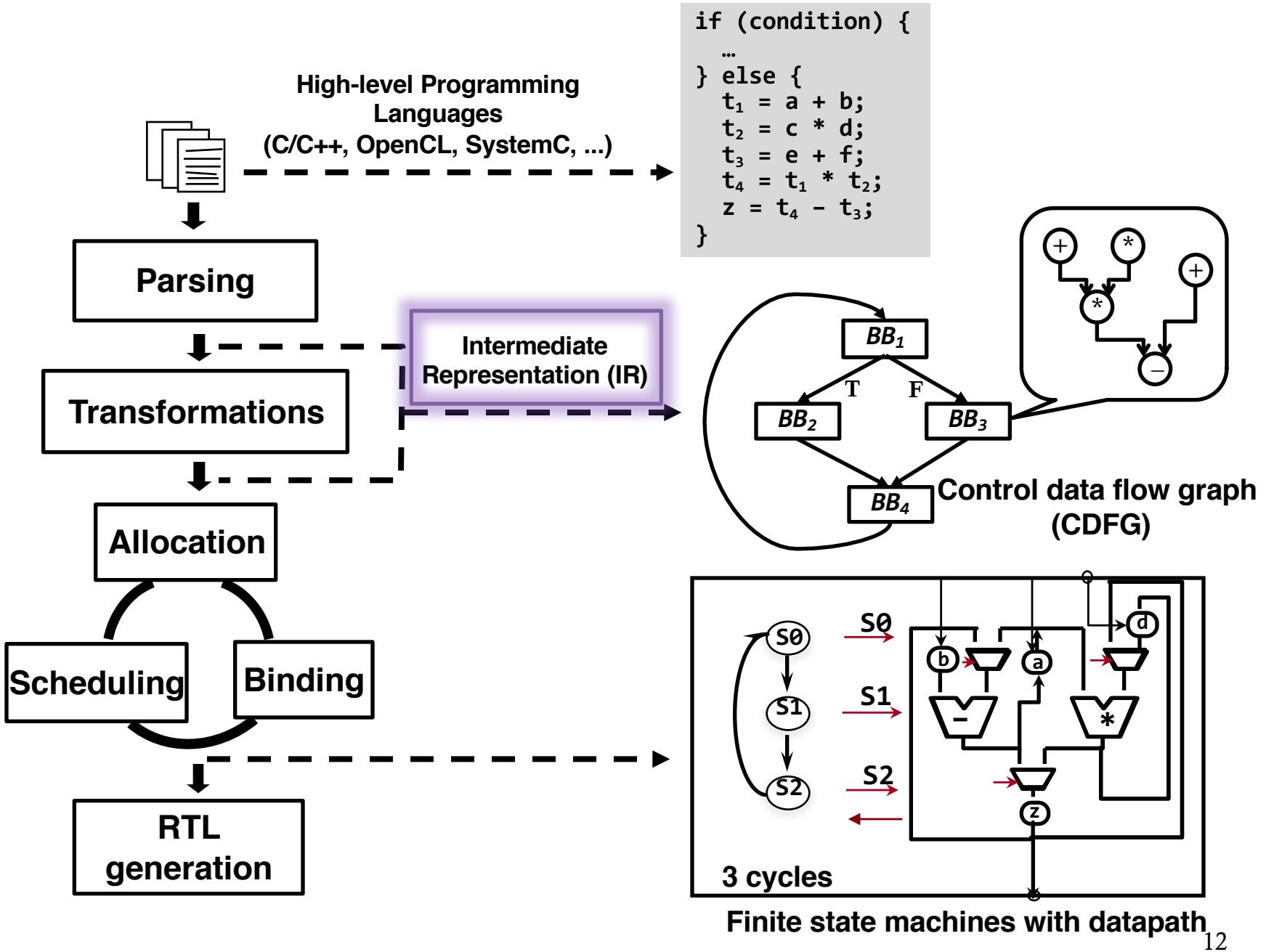
- Assumptions
  - K=3
  - All inputs arrive at time 0
  - Unit delay model: 3-LUT delay = 1; Zero edge delay
- Question: **Minimum arrival time** (AT) of each gate output?



FPGA Design Flow with HLS



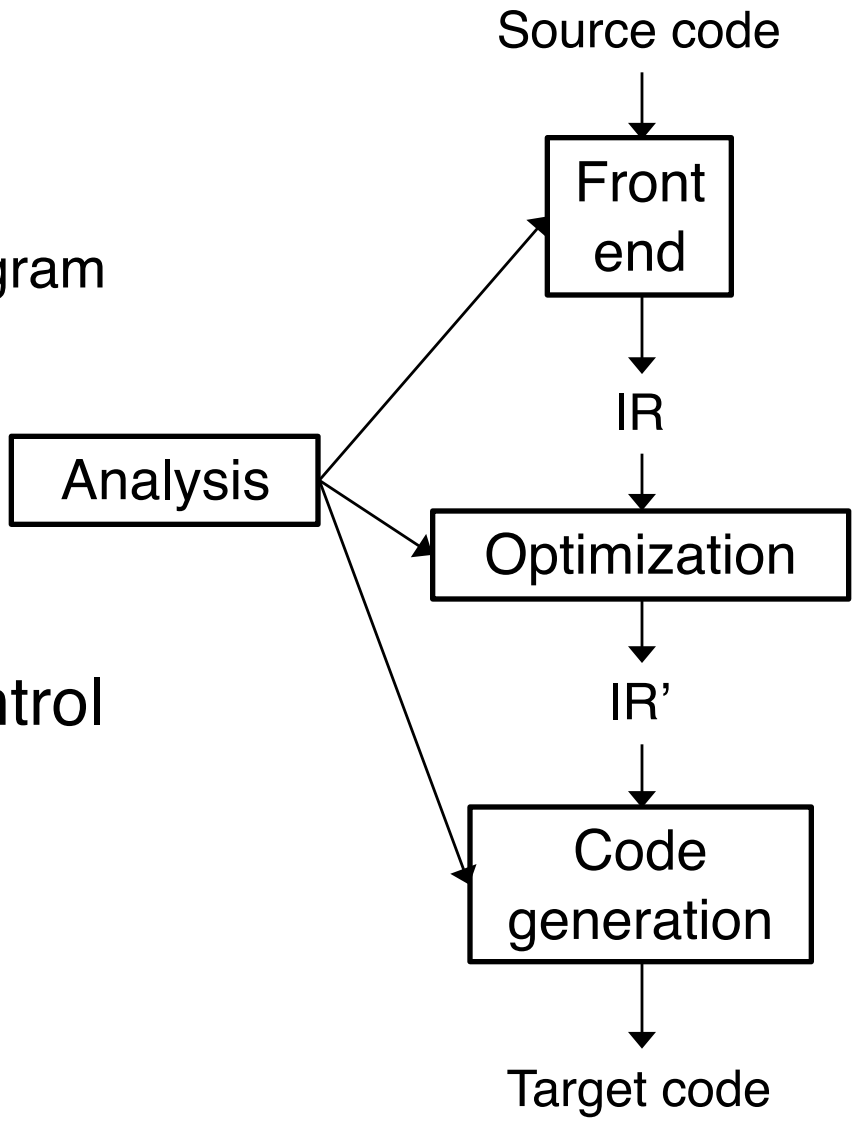
A Typical HLS Flow



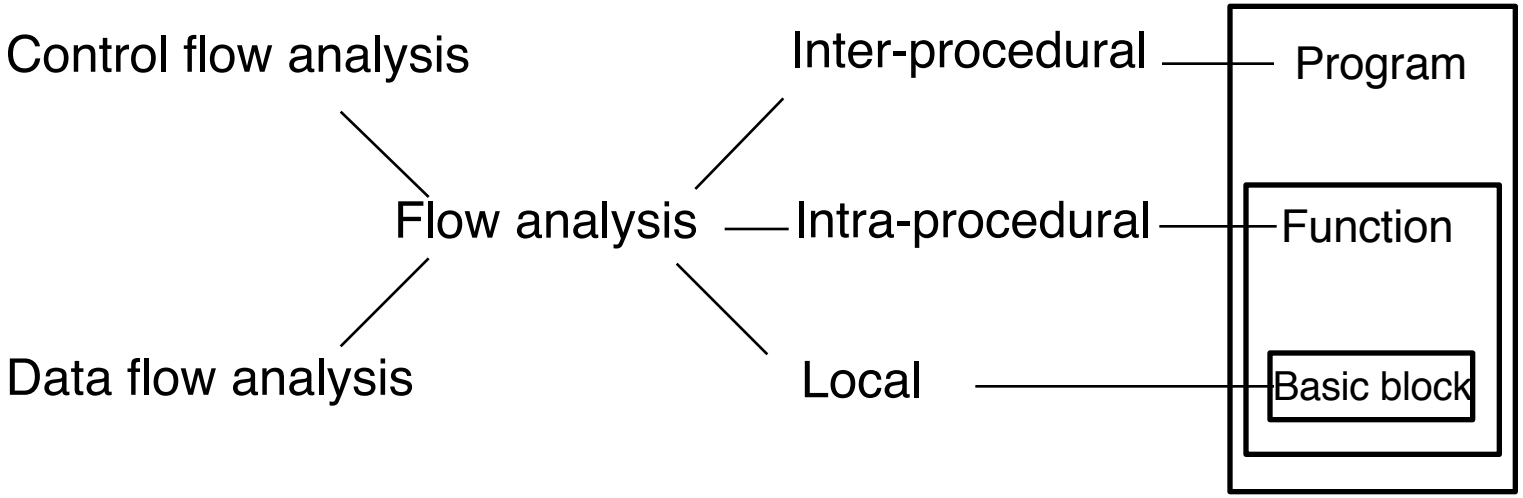
Intermediate Representation (IR)

- Purposes of creating and operating on an IR
  - Encode the behavior of the program
  - Facilitate analysis
  - Facilitate optimization
  - Facilitate retargeting

- The IR we will focus on is control data flow graph (CDFG)



Program Flow Analysis



- Control flow analysis: determine control structure of a program and build control flow graphs (**CFGs**)
- Data flow analysis: determine the flow of data values and build data flow graphs (**DFGs**)

Basic Blocks

- **Basic block**: a sequence of consecutive intermediate language statements in which flow of control can only enter at the beginning and leave at the end
  - Only the last statement of a basic block can be a branch statement and only the first statement of a basic block can be a target of a branch

Partitioning a Program into Basic Blocks

- Each basic block begins with a leader statement
- Identify leader statements (i.e., the first statements of basic blocks) by using the following rules:
  - (i) The **first statement** in the program is a leader
  - (ii) Any statement that is the **target of a branch statement** is a leader (for most intermediate languages these are statements with an associated label)
  - (iii) Any statement that **immediately follows a branch** or **return** statement is a leader

Example: Forming the Basic Blocks

```
(1)   p = 0
(2)   i = 1
(3)   t1 = 4 * i
(4)   t2 = a[t1]
(5)   t3 = 4 * i
(6)   t4 = b[t3]
(7)   t5 = t2 * t4
(8)   t6 = p + t5
(9)   p = t6
(10)  t7 = i + 1
(11)  i = t7
(12)  if i <= 20 goto (3)
(13)  j = …
```

Leader statement is:
(1) the first in the program
(2) any that is the target of a branch
(3) any that immediately follows a branch

Basic Blocks:

B1
```
(1) p = 0
(2) i = 1
```

B2
```
(3)   t1 = 4 * i
(4)   t2 = a[t1]
(5)   t3 = 4 * i
(6)   t4 = b[t3]
(7)   t5 = t2 * t4
(8)   t6 = p + t5
(9)   p = t6
(10)  t7 = i + 1
(11)  i = t7
(12)  if i <= 20 goto (3)
```

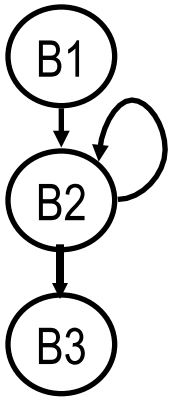
B3
```
(13) j = …
```

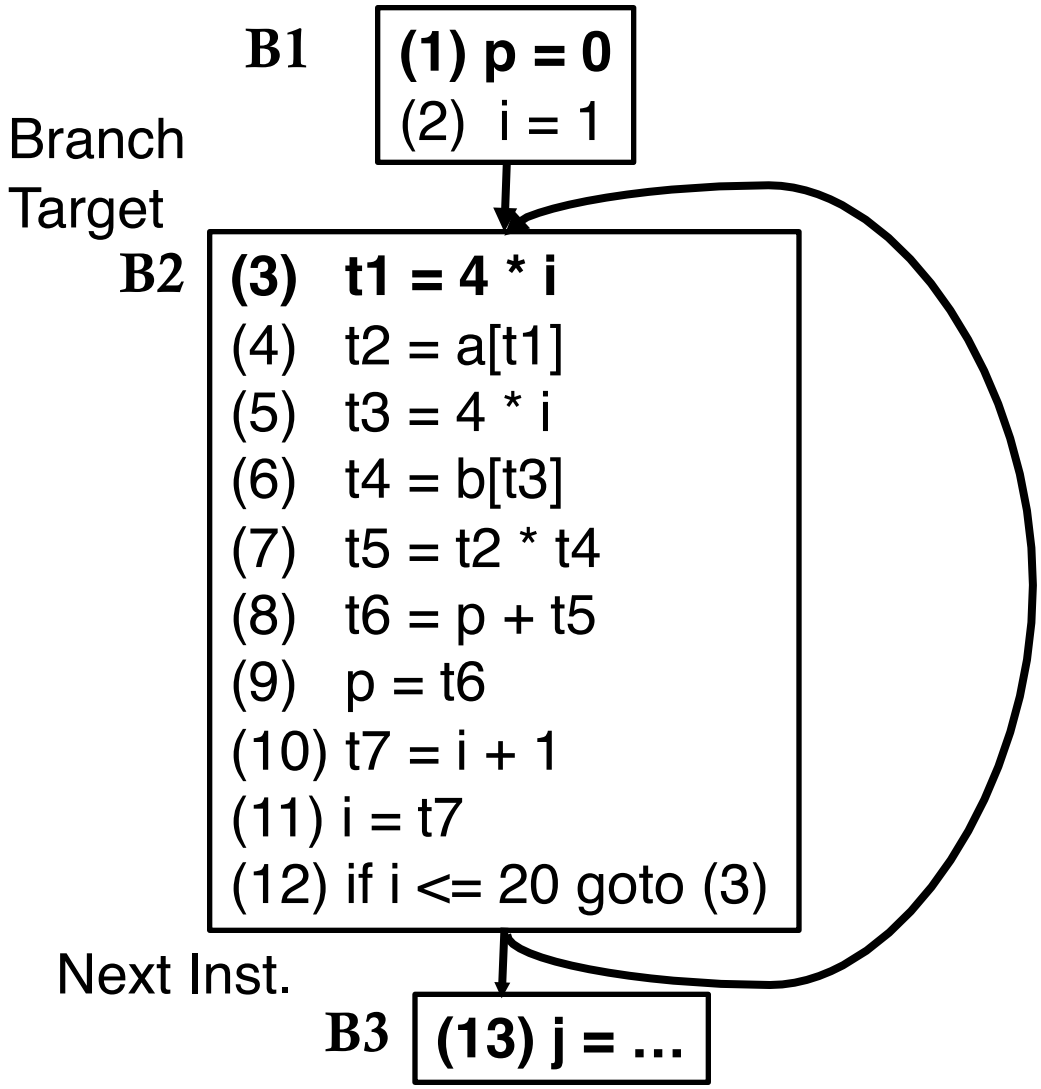
Control Flow Graph (CFG)

- A **control flow graph** (CFG), or simply a flow graph, is a directed graph in which:
  - (i) the nodes are basic blocks; and
  - (ii) the edges are induced from the possible flow of the program
- The basic block whose leader is the first intermediate language statement is called the **entry node**
- In a CFG we assume no information about data values
  - an edge in the CFG means that the program **may** take that path

Example: Control Flow Graph Formation

CFG





Dominators

- A node *p* in a CFG ***dominates*** a node *q* if every path from the entry node to *q* goes through *p*. We say that node *p* is a ***dominator*** of node *q*

- The ***dominator set*** of node *q*, **DOM(*q*),** is formed by all nodes that dominate *q*
  - By definition, each node dominates itself therefore, $q \in$ **DOM(*q*)**

Dominance Relation

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

  N: set of nodes

  E: set of edges

  s: entry node and

  let $p \in N$, $q \in N$

  - p **dominates** q, written $p \leq q$
    - $p \in \text{DOM}(q)$
  - 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 $t \in N$ such that $p < t < q$
    - $p = \text{IDOM}(q)$

Example: Dominance Relation

- **Dominator sets:**

  DOM(1) = {1}

  DOM(2) = {1, 2}

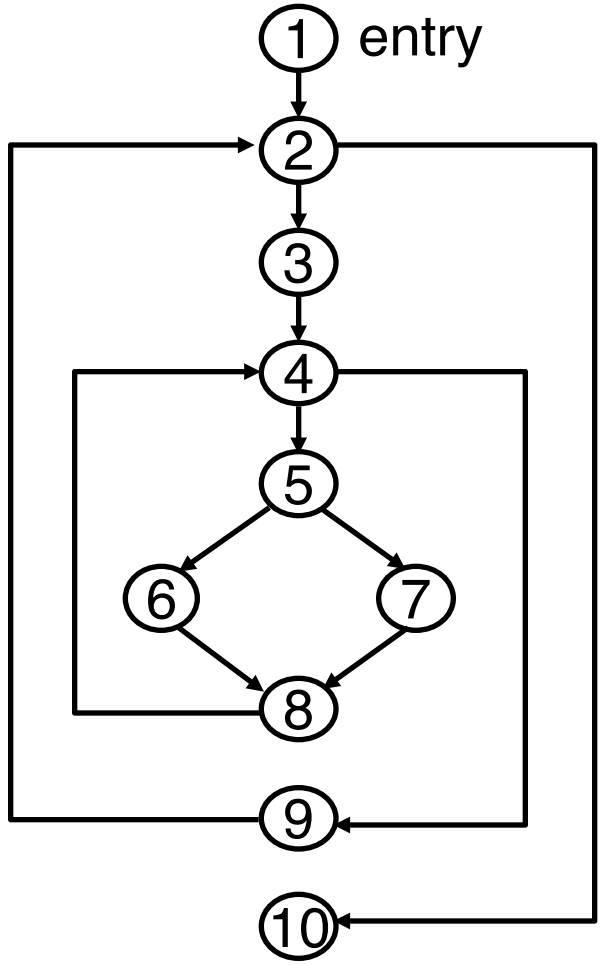
  DOM(3) = {1, 2, 3}

  DOM(10) = {1, 2, 10}

- **Immediate domination:**

  $1 <_d 2$, $2 <_d 3$, …

  IDOM(2) = 1, IDOM(3) = 2 …

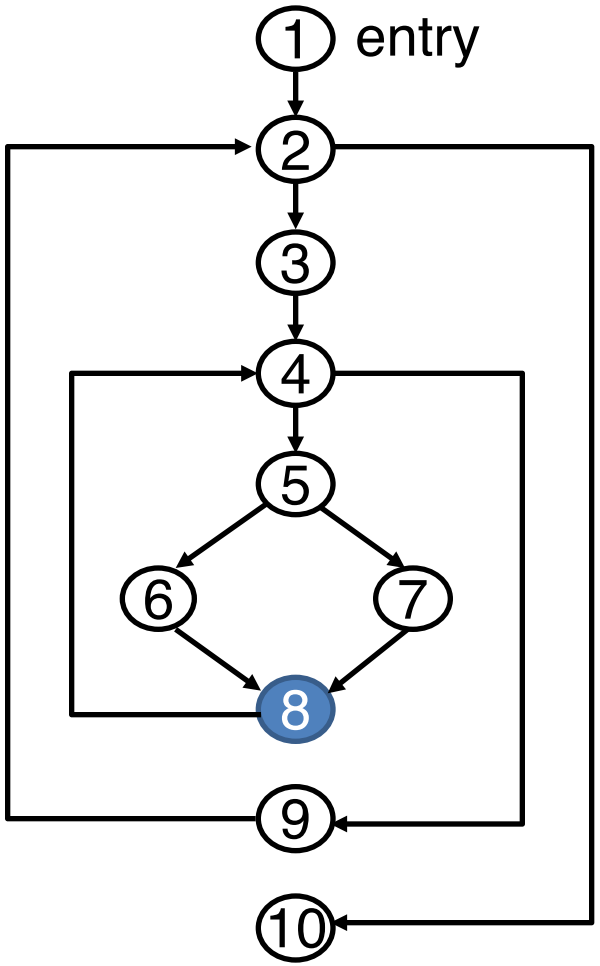


Dominance Question

Assume that node *P* is an immediate dominator of node *Q*

Question: Is *P* necessarily an immediate predecessor of *Q* in the CFG?

Answer: **NO**



Identifying Loops

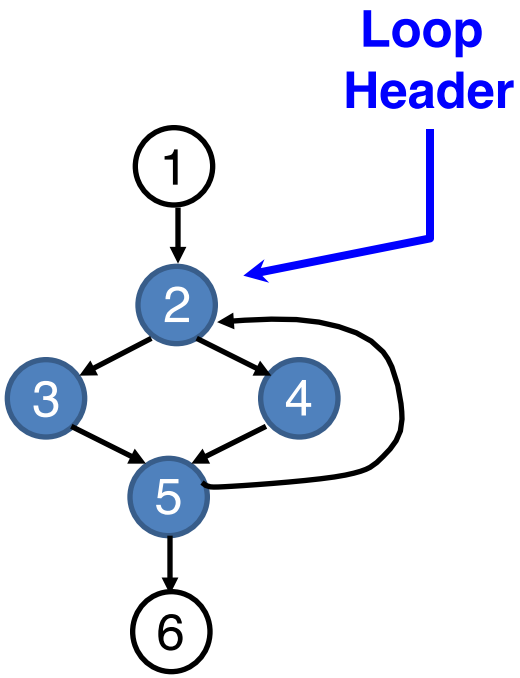
- **Motivation**: Programs spend most of the execution time in loops, therefore there is a larger payoff for optimizations that exploit loop structure

- **Goal**: Identify loops in a CFG, not sensitive to input syntax
  - Create an uniform treatment for program loops written using different syntactical constructs (e.g. while, for, goto)

- **Approach:** Use a general approach based on analyzing graph-theoretical properties of the CFG

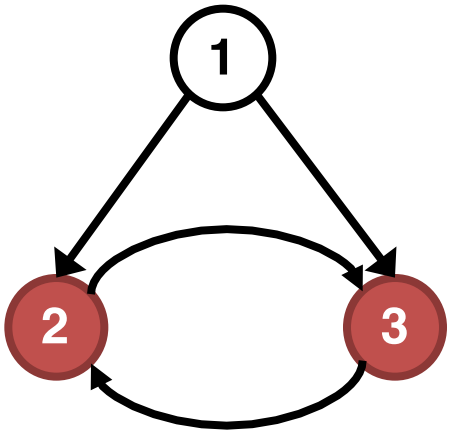
Loop Definition

- Definition: A **strongly connected component** (SCC) of the CFG, with
  - a single entry point called the **header** which dominates all nodes in the SCC



Is it a Loop?

- Question: In the CFG below, nodes 2 and 3 form an SCC; but **do they form a loop**?



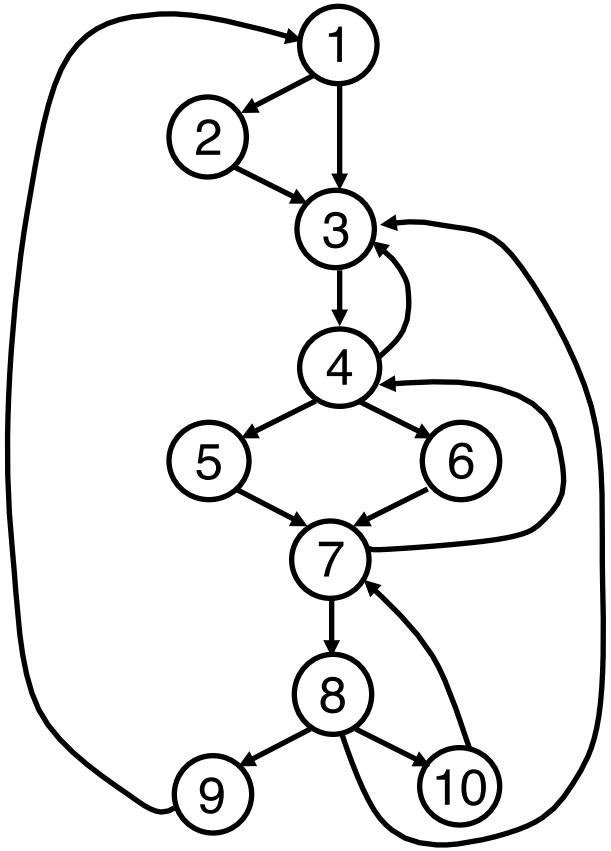
No header
node 2 does Not dominate 3;
Neither does node 3

nodes 2 and 3 form an SCC

Finding Loops

- Loop identification algorithm
  - Find an edge $B \rightarrow H$ where H dominates B; This edge is called a **back-edge**
  - Find all nodes that (1) are dominated by H and (2) can reach B through nodes dominated by H; add them to the loop
    - H and B are naturally included

Finding Loops



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

Summary

- Basic Blocks
  - Group of statements that execute atomically
- Control Flow Graphs
  - Model the control dependences between basic blocks
- Dominance relations
  - Shows control dependences between BBs
  - Used to determine natural loops

Next Class

- Next lecture: Static single assignment

Acknowledgements

- These slides contain/adapt materials developed by
  - Forrest Brewer (UCSB)
  - Ryan Kastner (UCSD)
  - Prof. José Amaral (Alberta)