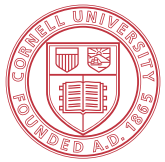
ECE 5775
High-Level Digital Design Automation
Fall 2018

Binary Decision Diagrams
Static Timing Analysis



Cornell University

CSL

Announcements

- Lab 1 (CORDIC design) due on Monday 11:59pm
  - Fixed-point design should not have usage of DSP48s
  - First TA office hour on Monday 11am-noon in Rhodes 312

- First problem set will be released on Monday

Review: Quantization for Fixed-Point Types

- Write down the **value of 'y' in decimal** after the assignment for each of the following cases
  - Assume truncation for quantization (AP_TRN), and wrap for overflow (AP_WRAP)

Binary

(1)

```
ap_fixed<4, 2>   x = -0.25;
ap_fixed<3, 1>   y = x;
```

x = 11.11 → y = 1.11

-0.125

(2)

```
ap_fixed<4, 2>   x = 0.5;
ap_fixed<3, 3>   y = x;
```

0

integer

no frac bits

Outline

- Basics of graph theory

- Graph algorithms applied to two EDA problems
  - Binary decision diagrams

> "**One of the only really fundamental data structures that came out in the last twenty-five years**" – Donald Knuth, 2008

  - Static timing analysis

Graph Definition

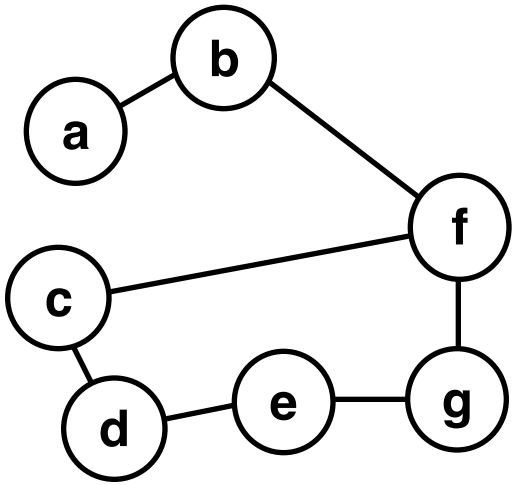
- Graph: a set of objects and their connections
  - Importance: any binary relation can be represented as a graph

- Formal definition:
  - $G = (V, E)$, $V = \{v_1, v_2, ..., v_n\}$, $E = \{e_1, e_2, ..., e_m\}$
    - V : set of **vertices** (nodes), E : set of **edges** (arcs)
  - **Undirected graph**: if an edge $\{u, v\}$ also implies $\{v, u\}$
  - **Directed graph**: each edge (u, v) has a direction

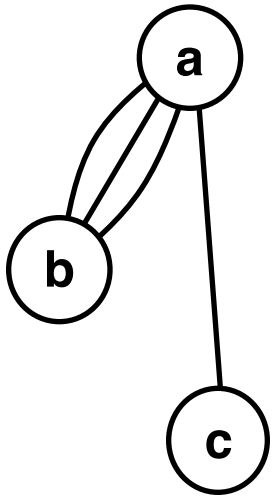
Simple Graph

- Loops, multi edges, and simple graphs
  - An edge of the form (*x*, *x*) is said to be a **self-loop**
  - A graph permitted to have multiple edges (or parallel edges) between two vertices is called a **multigraph**
  - A graph is said to be **simple** if it contains no self-loops or multiedges

Simple graph



Multigraph



Graph Connectivity

- Paths
  - A **path** is a sequence of edges connecting two vertices
  - A **simple path** never goes through any vertex more than once

- Connectivity
  - A graph is **connected** if there is there is a path between any two vertices
  - Any subgraph that is connected can be referred to as a **connected component**
  - A directed graph is **strongly connected** if there is always a directed path between vertices

Trees and DAGs

- A **cycle** is a path starting and ending at the same vertex. A cycle in which no vertex is repeated other than the starting vertex is said to be a **simple cycle**

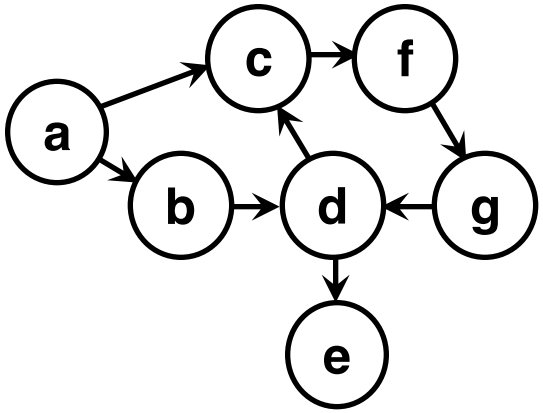
A tree with N vertices has N-1 edges

- An undirected graph with no cycles is a **tree** if it is connected, or a **forest** otherwise
  - A **directed tree** is a directed graph which would be a tree if the directions on the edges were ignored

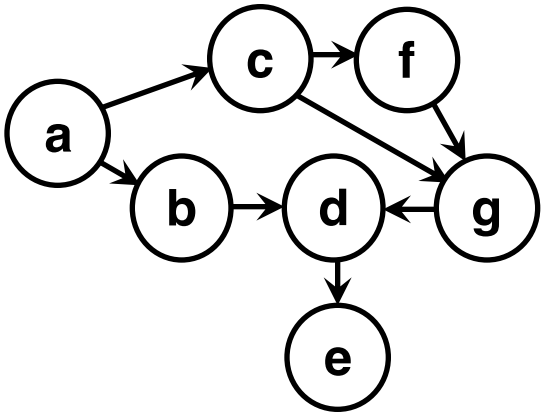
- A directed graph with no directed cycles is said to be a **directed acyclic graph** (**DAG**)

Examples

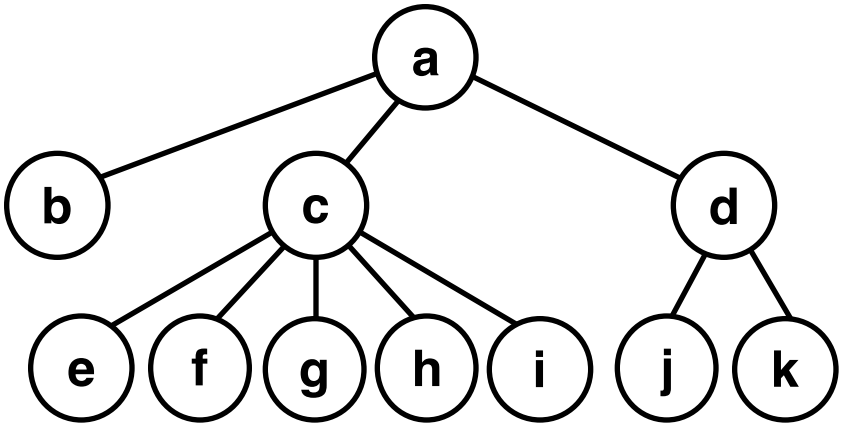
Directed graphs with cycles



Directed acyclic graph (DAG)

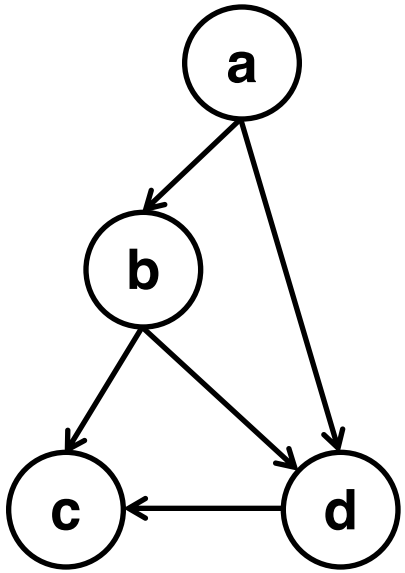


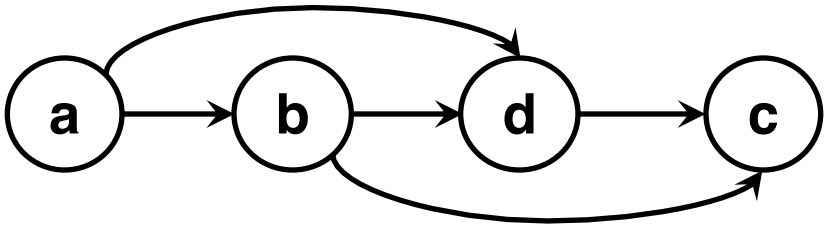
Tree



Topological Sort

- A **topological sort** (or order) of a directed graph is an ordering of nodes where all edges go from an earlier vertex (left) to a later vertex (right)
  - Feasible if and only if the subject graph is a DAG





Binary Decision Diagrams

IEEE TRANSACTIONS ON COMPUTERS, VOL. C-35, NO. 8, AUGUST 1986

Graph-Based Algorithms for Boolean Function Manipulation

RANDAL E. BRYANT, MEMBER, IEEE

Abstract—**In this paper we present a new data structure for representing Boolean functions and an associated set of manipulation algorithms. Functions are represented by directed, acyclic graphs in a manner similar to the representations introduced by Lee [1] and Akers [2], but with further restrictions on the ordering of decision variables in the graph. Although a function requires, in the worst case, a graph of size exponential in the number of arguments, many of the functions encountered in typical applications have a more reasonable representation. Our algorithms have time complexity proportional to the sizes of the graphs being operated on, and hence are quite efficient as long as the graphs do not grow too large. We present experimental results from applying these algorithms to problems in logic design verification that demonstrate the practicality of our approach.**

Index Terms—**Boolean functions, binary decision diagrams, logic design verification, symbolic manipulation.**

I. Introduction

BOOLEAN Algebra forms a cornerstone of computer science and digital system design. Many problems in digital logic design and testing, artificial intelligence, and combinatorics can be expressed as a sequence of operations on Boolean functions. Such applications would benefit from efficient algorithms for representing and manipulating Boolean functions symbolically. Unfortunately, many of the tasks one would like to perform with Boolean functions, such as testing whether there exists any assignment of input variables such that a given Boolean expression evaluates to 1 (satisfiability), or two Boolean expressions denote the same function (equivalence) require solutions to NP-complete or co NP-complete problems [3]. Consequently, all known approaches to performing these operations require, in the worst case, an amount of computer time that grows exponentially with the size of the problem. This makes it difficult to compare the relative efficiencies of different approaches to representing and manipulating Boolean functions. In the worst case, all known approaches perform as poorly as the naive approach of representing functions by their truth tables and defining all of the desired operations in terms of their effect on truth table entries. In practice, by utilizing more clever representations and manipulation algorithms, we can often avoid these exponential computations.

Manuscript received November 28, 1984; revised June 11, 1985. This work was supported in part by the Defense Advanced Research Projects Agency under Orders 3771 and 3597.

The author is with the Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA 15213.

IEEE Log Number 8609399.

A variety of methods have been developed for representing and manipulating Boolean functions. Those based on classical representations such as truth tables, Karnaugh maps, or canonical sum-of-products form [4] are quite impractical—*every* function of n arguments has a representation of size 2^n or more. More practical approaches utilize representations that, at least for many functions, are not of exponential size. Example representations include as a reduced sum of products [4] (or equivalently as sets of prime cubes [5]) and factored into unate functions [6]. These representations suffer from several drawbacks. First, certain common functions still require representations of exponential size. For example, the even and odd parity functions serve as worst case examples in all of these representations. Second, while a certain function may have a reasonable representation, performing a simple operation such as complementation could yield a function with an exponential representation. Finally, none of these representations are *canonical forms,* i.e., a given function may have many different representations. Consequently, testing for equivalence or satisfiability can be quite difficult.

Due to these characteristics, most programs that process a sequence of operations on Boolean functions have rather erratic behavior. They proceed at a reasonable pace, but then suddenly "blow up," either running out of storage or failing to complete an operation in a reasonable amount of time.

In this paper we present a new class of algorithms for manipulating Boolean functions represented as directed acyclic graphs. Our representation resembles the binary decision diagram notation introduced by Lee [1] and further popularized by Akers [2]. However, we place further restrictions on the ordering of decision variables in the vertices. These restrictions enable the development of algorithms for manipulating the representations in a more efficient manner.

Our representation has several advantages over previous approaches to Boolean function manipulation. First, most commonly encountered functions have a reasonable representation. For example, all symmetric functions (including even and odd parity) are represented by graphs where the number of vertices grows at most as the square of the number of arguments. Second, the performance of a program based on our algorithms when processing a sequence of operations degrades slowly, if at all. That is, the time complexity of any single operation is bounded by the product of the graph sizes for the functions being operated on. For example, complementing a function requires time proportional to the size of the function graph, while combining two functions with a binary operation (of which intersection, subtraction, and testing for

0018-9340/86/0800-0677$01.00

One of the most cited papers in CS/CE

Ideal Representation of a Boolean Function

- We wish to find a representation with the following characteristics
  - **Compact** (in terms of size)
  - **Efficient** to compute the output with the given inputs and efficient to manipulate and modify
  - Ideally, a **canonical** representation
    - Equivalent functions have the same unique form (under certain restrictions)

Example: Voting Function

- A Boolean voting function
  - An *n*-ary Boolean function $f(x_1, x_2, ..., x_n)$ evaluates to 1 if 50% or more ($\geq \lceil n/2 \rceil$) of its inputs are set to 1
  - Examples:
    - f(0,0) = 0
    - f(0,1) = 1
    - f(0,0,1) = 0
    - f(1,0,1) = 1

- How to formally represent this function?
  - Truth table
  - Karnaugh map
  - Sum of Products (SOP)
  - …

Truth Table

x	y	z	f
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

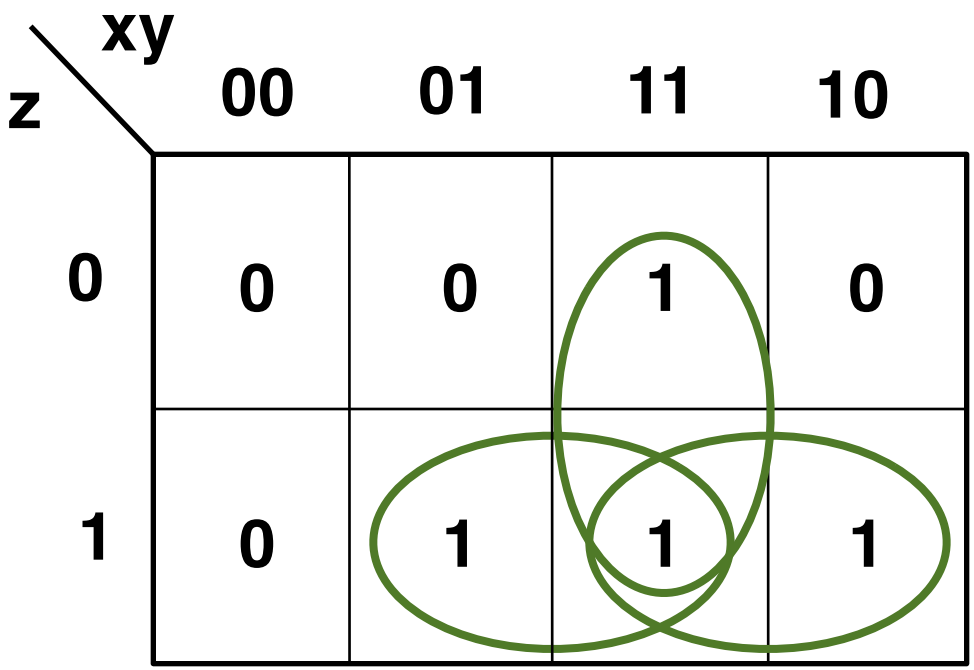
Truth table is canonical

But 2^n table entries are required!

Canonical sum (4 minterms):

xyz' + xy'z + xyz + x'yz

Karnaugh Map and SOP



Minimized SOP (3 terms): xy + xz + yz

What about *n* inputs? (esp. where *n* is large)

Note: K-map only handles up to 6 inputs;
The output is not necessarily unique

Complexity of SOP Representation

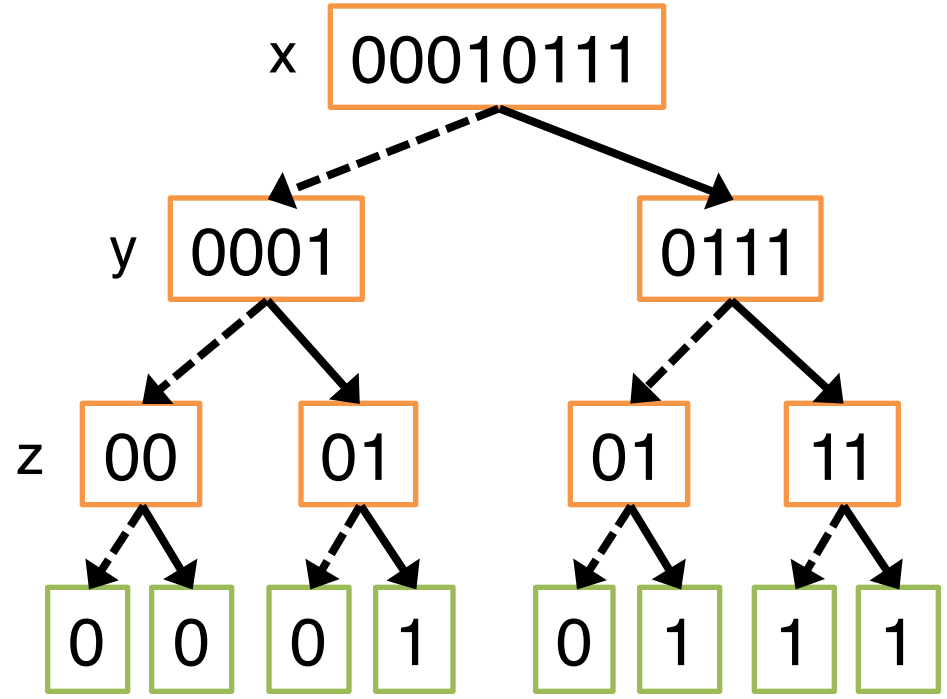
- An n-input Boolean voting function has at least C(n, n/2) prime implicants

- Growth rate of C(*n*, *k*) in terms of *n*
  - For *k*=1, C(*n*,1) = *n*
  - For *k*=2, C(*n*,2) = *n*(*n*-1)/2
  - For *k*=3, C(*n*,3) = *n*(*n*-1)(*n*-2)/6
  - …
  - For k=*n*/2, C(*n*, *n*/2) = $\frac{n!}{[(n/2)!]^2} \in \Theta(2^n n^{-0.5})$ (involves Stirling formula)

Truth Table, Shannon Expansion, and Decision Tree

	x	y	z	f
x=0, y=0	0	0	0	0
	0	0	1	0
x=0, y=1	0	1	0	0
	0	1	1	1
x=1	1	0	0	0
	1	0	1	1
	1	1	0	1
	1	1	1	1

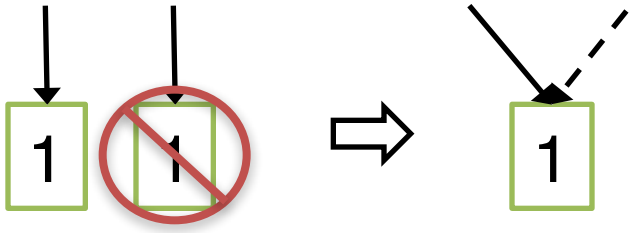
$$f = x' \bullet f_{x=0} + x \bullet f_{x=1}$$

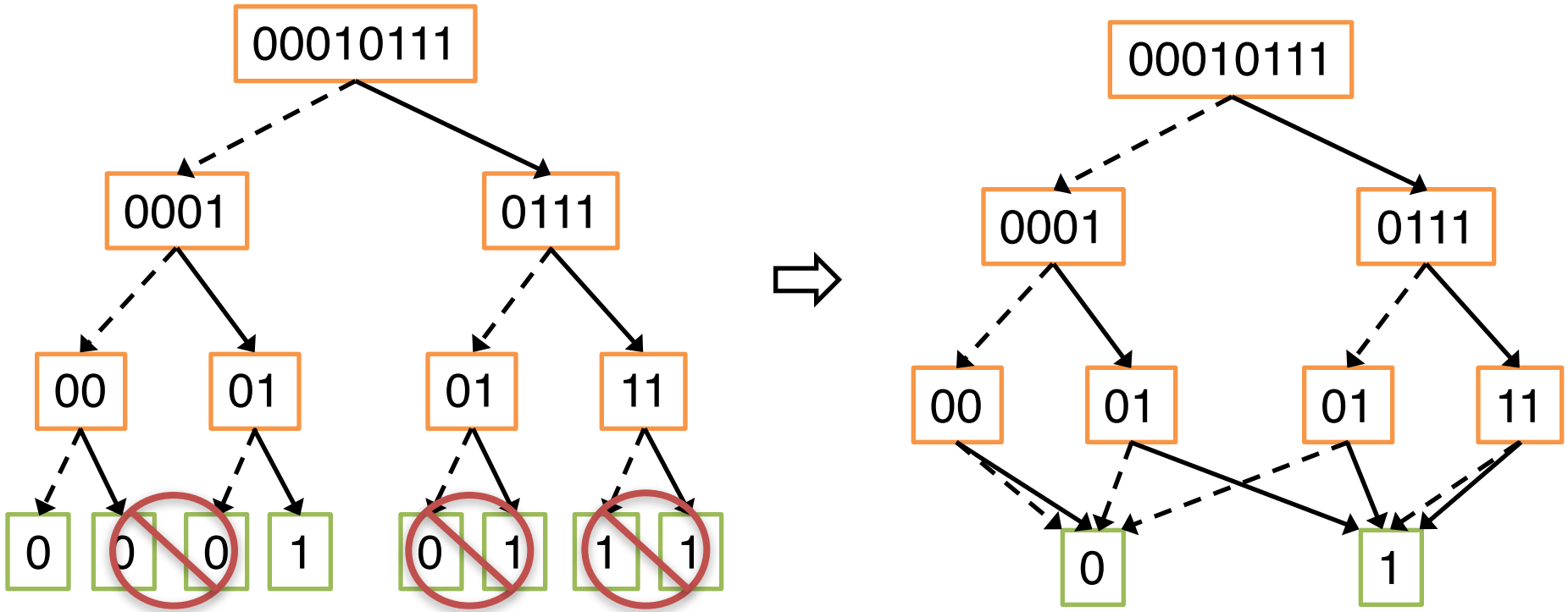


- Nonterminal node in orange
  - Follow dashed line for value 0
  - Follow solid line for value 1
- Terminal (leaf) node in green
  - Function value determined by leaf values

Reduction Rule #1

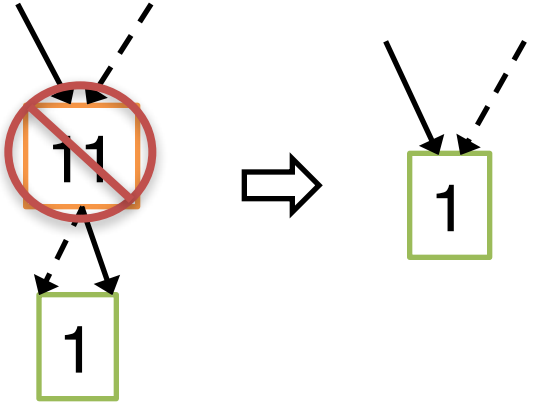
- Merge equivalent leaves

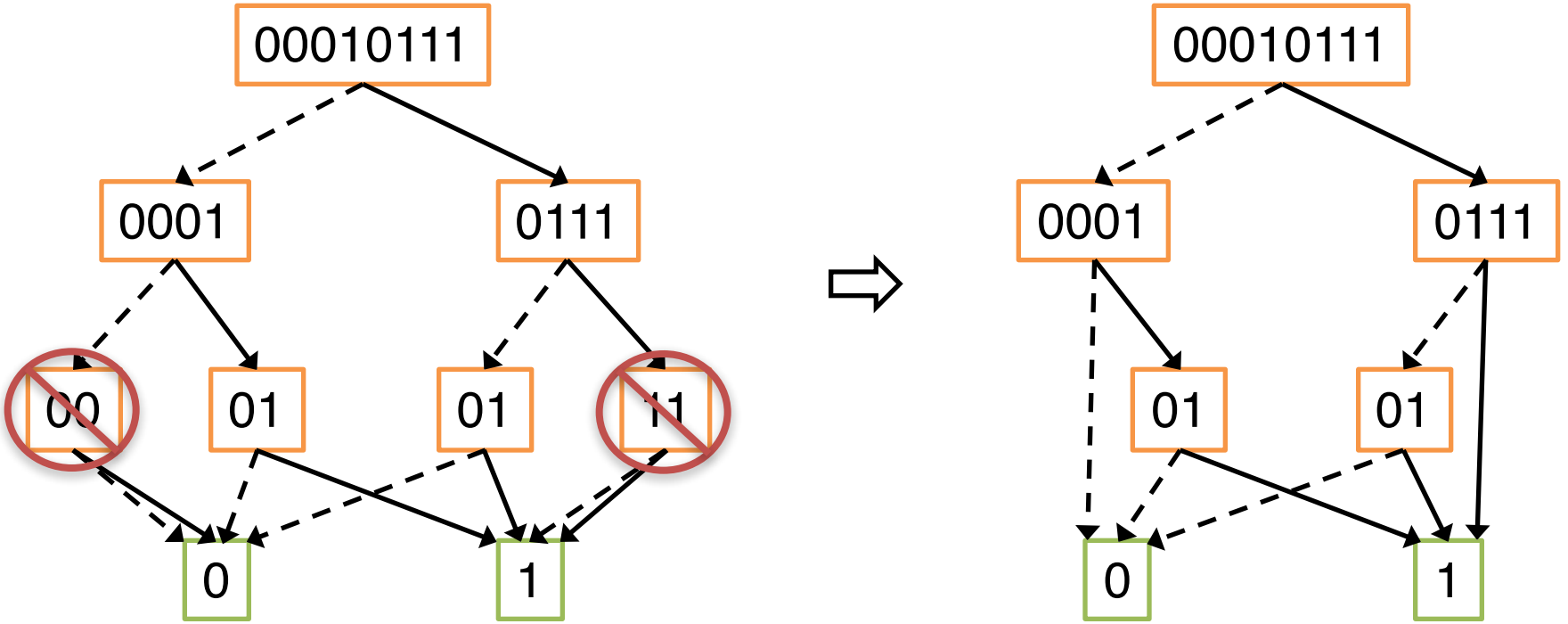




Reduction Rule #2

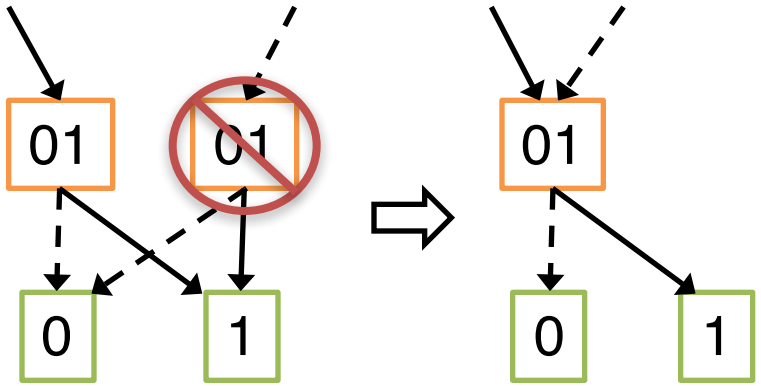
- Remove redundant tests
  - For a node v, left(v) = right(v)

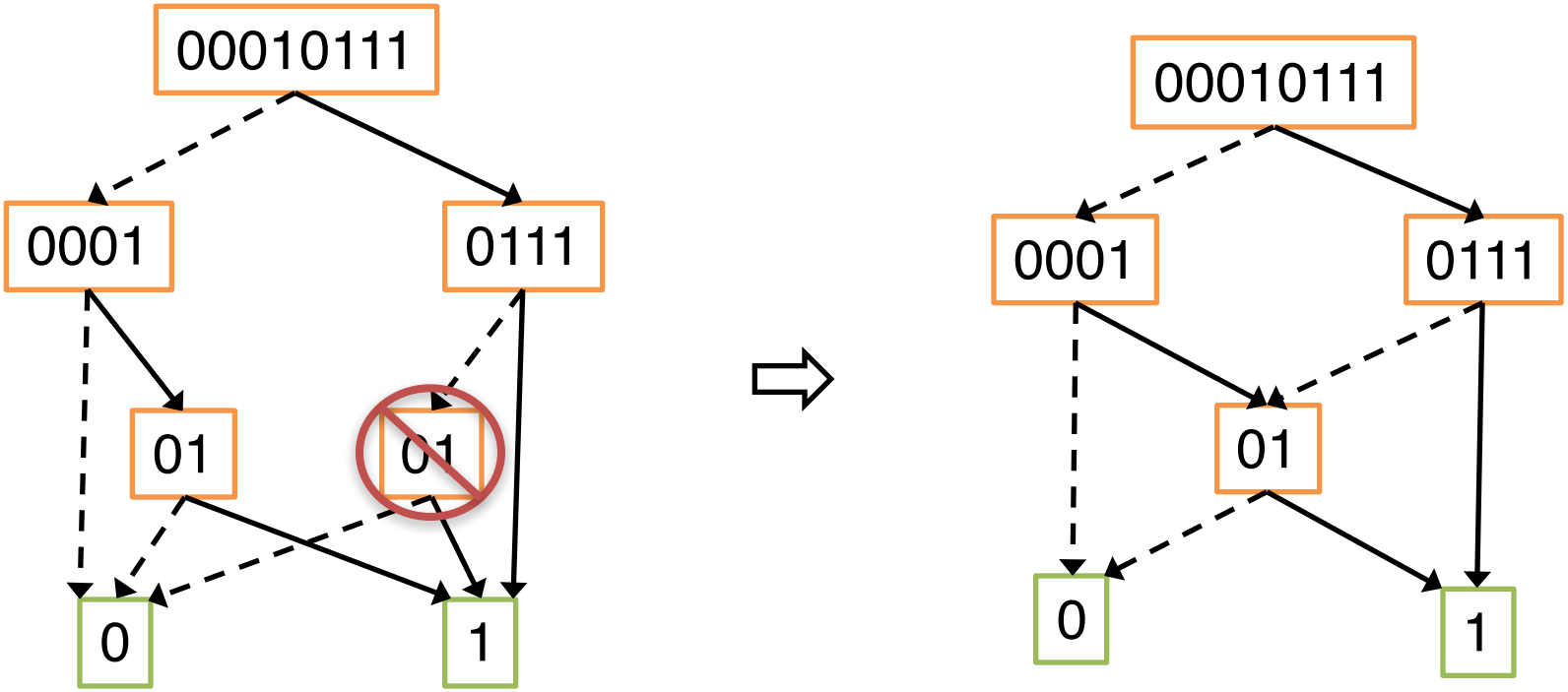




Reduction Rule #3

- Merge isomorphic nodes
  - u and v are isomorphic, when left(u) = left(v) and right(u) = right(v)





BDD Construction

- BDDs are usually directly constructed bottom up, avoiding the reduction steps

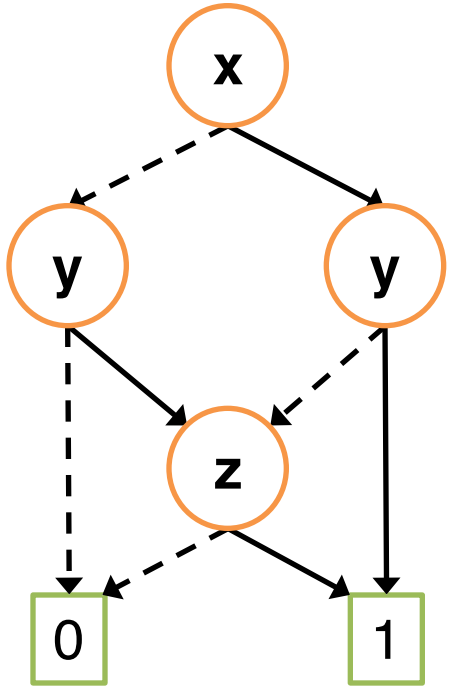
- One approach is using a hash table called unique table, which contains the IDs of the Boolean functions whose BDDs have been constructed
  - A new function is added if its associated ID is not already in the unique table

BDDs History

- Initially proposed by Lee in 1959, and later Akers in 1976
  - Idea of representing Boolean function as a rooted DAG with a decision at each vertex
- Popularized by Bryant in 1986
  - Further restrictions + efficient algorithms to make a useful data structure (ROBDD)
  - **BDD = ROBDD since then**

ROBDDs

- Reduced and Ordered (**RO**BDD)
  - Directed acyclic graph (DAG)
    - Two children per node
    - Two terminals 0, 1
  - **Ordered**:
    - Co-factoring variables (splitting variables) always follow the same order along all paths $x_1 < x_2 < x_3 < \ldots < x_n$
  - **Reduced:**
    - Any node with two identical children is removed (rule #2)
    - Two nodes with isomorphic BDDs are merged (rules #1 and #3)

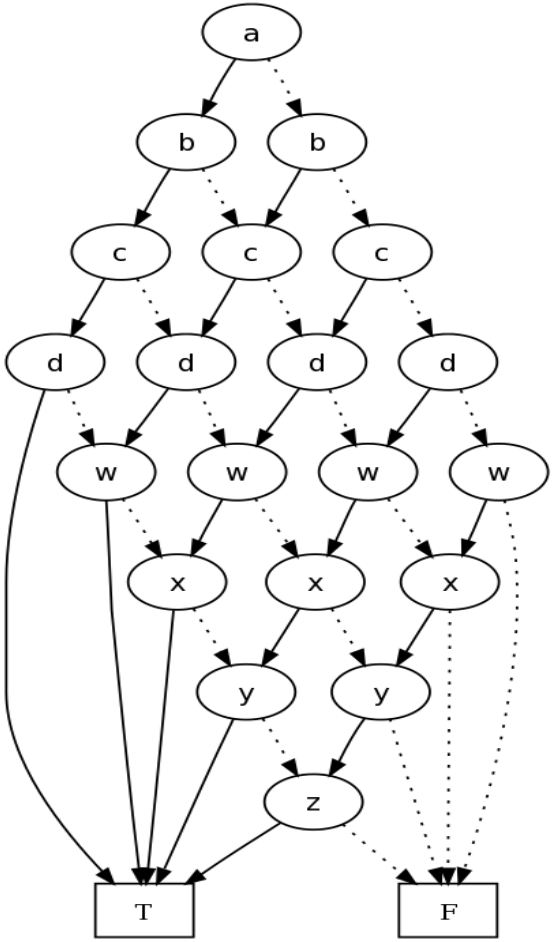


3-input voting function in BDD form

Canonical Representation

- BDD is a canonical representation of Boolean functions
  - Given the same variable order, two functions equivalent if and only if they have the same BDD form
    - “0” unique unsatisifable function
    - “1” unique tautology

BDD Representation of Voting Function

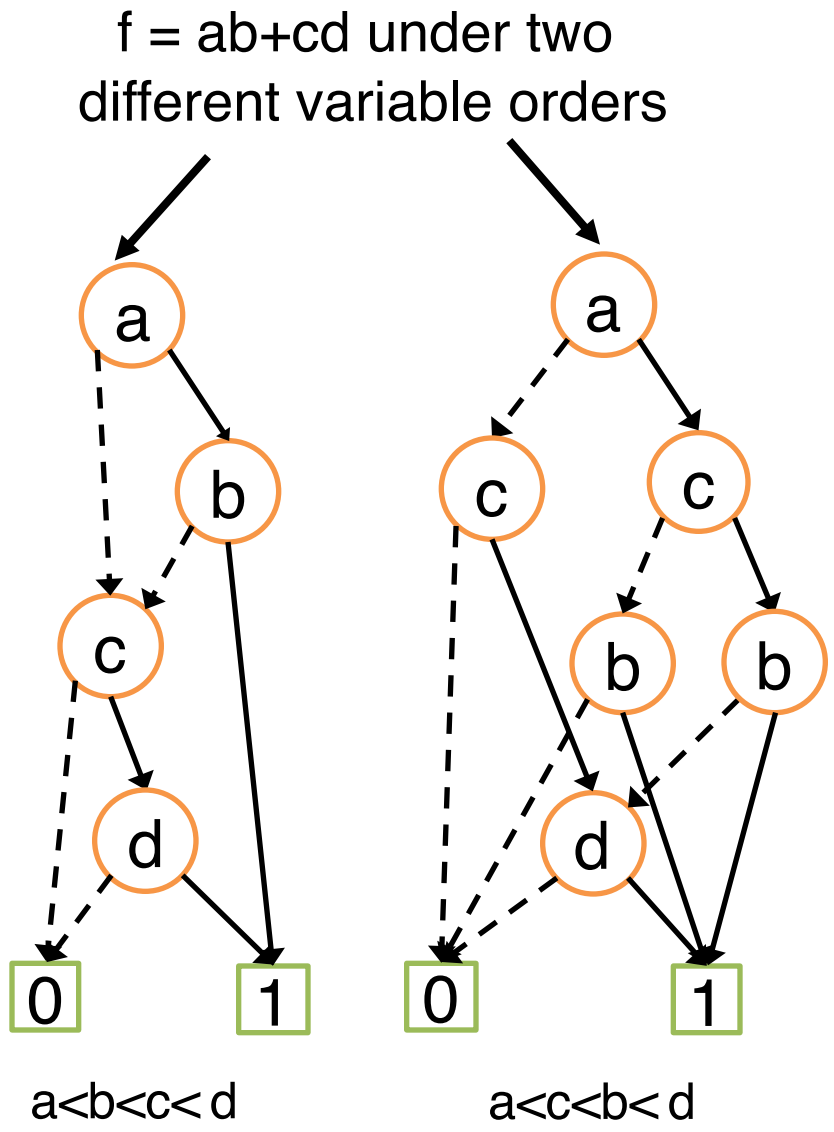


- 8-input voting function in BDD with only 20 nonterminal nodes
- In contrast to 70 prime implicants in SOP form

Diagram generated by www.cs.uc.edu/~weaversa/BDD_Visualizer.html

BDD Limitations

- NP-hard problem to construct the optimal order for a given BDD
- No efficient BDD exists for some functions regardless of the order
- Existing heuristics work reasonably well on many combinational functions from real circuits
  - Lots of research in ordering algorithms



Same function, two different orderings, different graphs

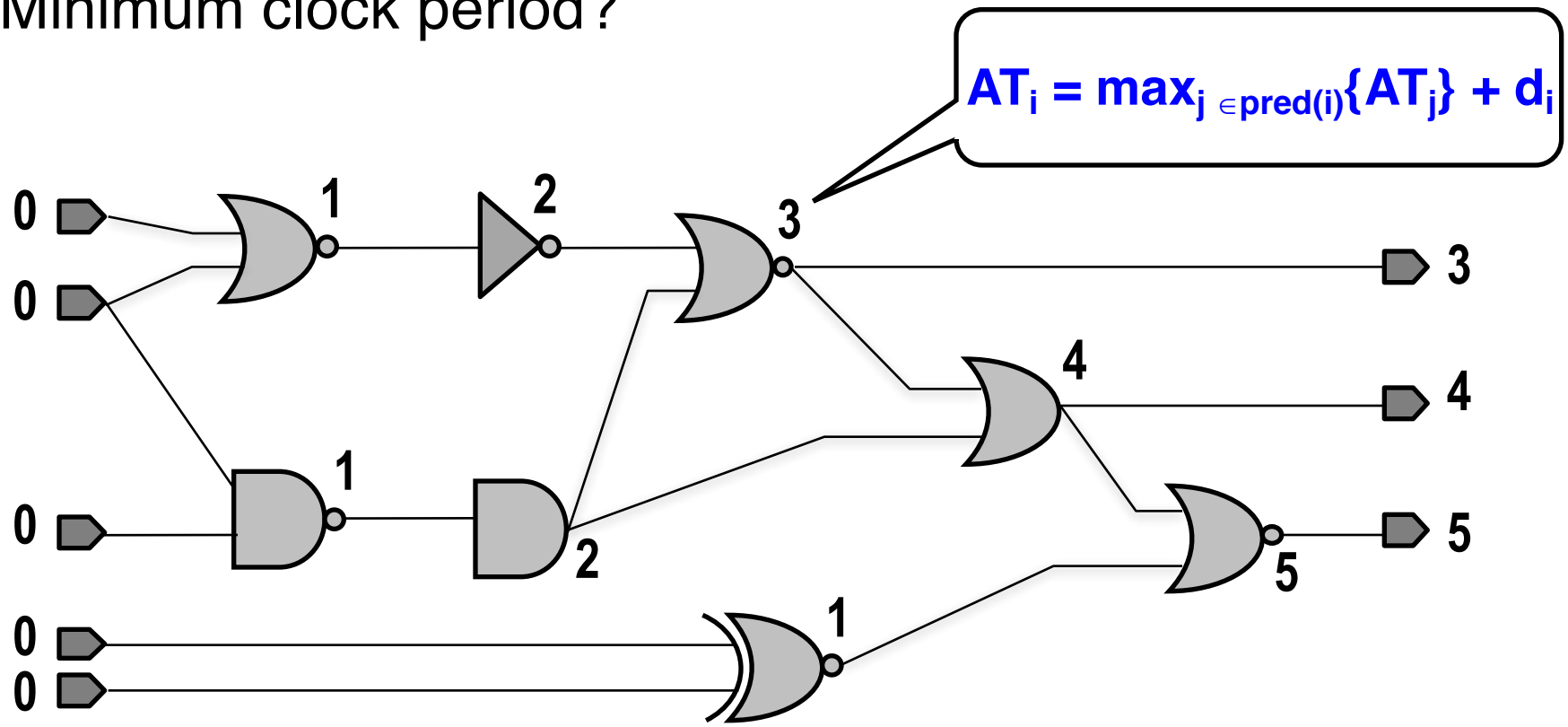
Static Timing Analysis

- In circuit graphs, **static timing analysis** (STA) refers to the problem of finding the delays from the input pins of the circuit (esp. nodes) to each gate
  - In sequential circuits, flip-flop (FF) input acts as output pin, FF output acts as input pin
  - Max delay of the output pins determines clock period
  - **Critical path** is a path with max delay among all paths

- Two important terms
  - **Required time**: The time that the data signal needs to arrive at certain endpoint on a path to ensure the timing is met
  - **Arrival time**: The time that the data signal actually arrives at certain endpoint on a path

STA: Arrival Times

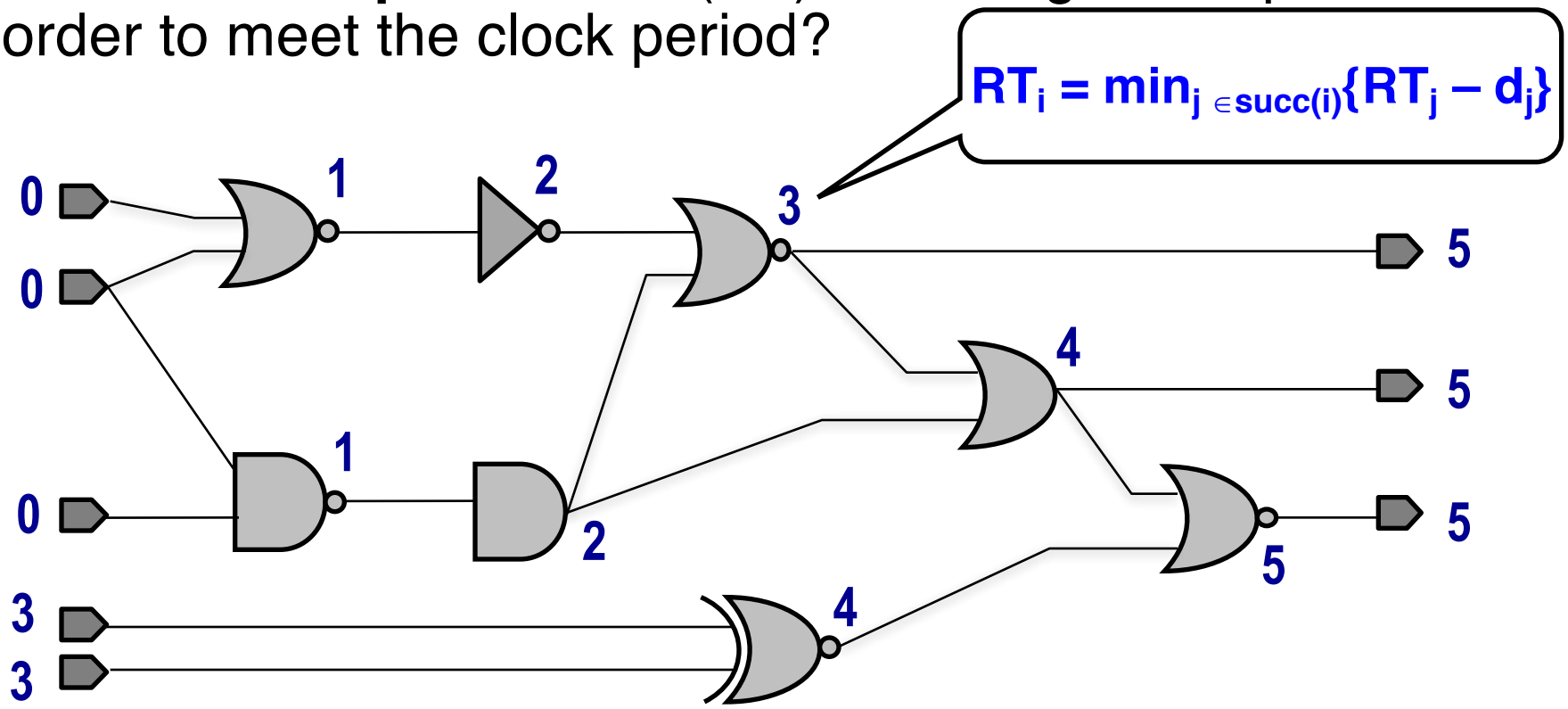
- Assumptions
  - All inputs arrive at time 0
  - All gate delays = 1ns (di = 1); all wire delays = 0
- Questions: **Arrival time** (AT) of each gate output? Minimum clock period?



Gates are visited in a topological order

STA: Required Times

- Assumptions
  - All inputs arrive at time 0
  - All gate delays = 1ns (di = 1); all wire delays = 0
  - Clock period = 5ns (200MHz frequency)
- Question: **Required time** (RT) of each gate output in order to meet the clock period?



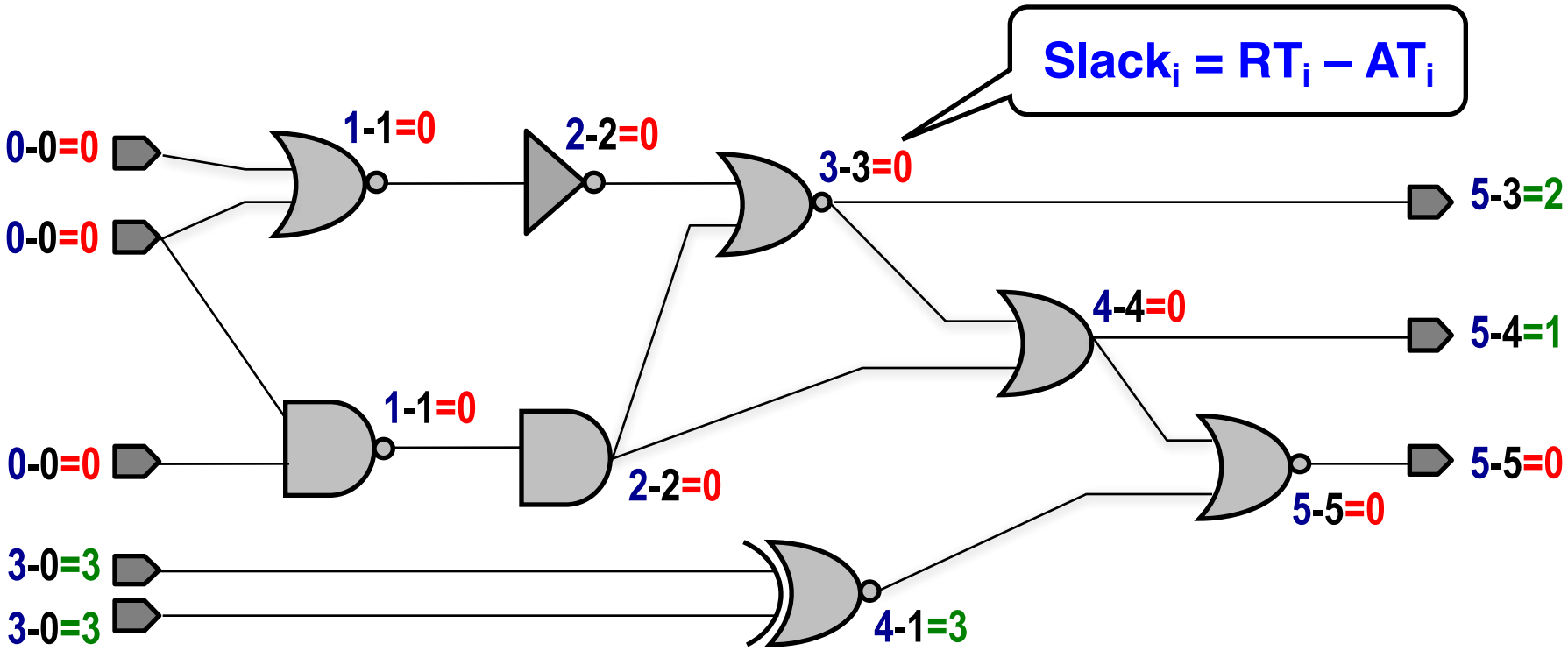
Gates are visited in a reverse topological order

More on Static Timing Analysis

- In addition to the arrival time and required time of each node, we are interested in knowing the **slack** (= RT - AT) of each node / edge
  - Negative slacks indicate unsatisfied timing constraints
  - Positive slacks often present opportunities for additional (area/power) optimization
  - Node on the **critical path** have zero slacks

STA: Slacks

- Assumptions:
  - All inputs arrive at time 0
  - All gate delays = 1ns, wire delay = 0
  - Clock period = 5ns
- Question: What is the maximum slowdown of each gate without violating timing?



Summary

- Graph algorithms are applicable to a wide range of EDA problems
  - Neatly capture the circuit topology
  - DAG is an important class of directed graph and will be used frequently in this class

Next Class

- Front-end compilation and CDFG

Acknowledgements

- These slides contain/adapt materials from / developed by
  - Prof. Randal Bryant (CMU)