Binary Decision Diagrams
Static Timing Analysis
Review: Graph Connectivity

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

- **Connectivity**
  - A graph is **connected** if 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**.

- 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
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

DFS order = a → ?

BFS order = a → ?
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
Agenda

- Graph algorithms applied to two EDA problems
  - Binary decision diagrams
  - Static timing analysis

Scientific Relevance (2005):

1. Optimization by Simulated Annealing, Kirkpatrick et al., 1983
2. Graph-Based Manipulation of Boolean Functions, Bryant, 1986

Most cited papers in Computer Science!!! (NEC, CiteSeer Database)

[source: Alberto Sangiovanni-Vincentelli’s keynote at ICCAD’2012]
Boolean Voting Function

- A Boolean voting function
  - An \( n \)-ary Boolean function \( f(x_1, x_2, \ldots, x_n) \) 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)
  ...

Ideally, we hope to find a representation with the following characteristics:

- **Compact** representation
- **Efficient** to compute the output with the given inputs
- Efficient to manipulate and modify
- **Canonical** representation
  - A unique form for equivalent functions under certain conditions
# Truth Table

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>z</th>
<th>f</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Truth table is canonical

But $2^n$ table entries are required!

Canonical SOP (4 minterms):

$$xyz’ + xy’z + xyz + x’y’z$$
Karnaugh Map and SOP

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

Note: K-map only handles up to 6 inputs

What about \( n \) inputs? (esp. where \( n \) is large)
Boolean Voting Function:
Exponential Growth Rate of SOP

- An n-input 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 $k=3$, $C(n,3) = n(n-1)(n-2)/6$
  - $\ldots$
  - For $k=n/2$, $C(n, n/2) = \frac{n!}{[(n/2)!]^2} \in \Theta(2^n n^{-0.5})$ (involves Stirling formula)
An Alternative: Binary Decision Diagrams

[Image of the page]

“One of the only really fundamental data structures that came out in the last twenty-five years” – Donald Knuth, 2008
Truth Table, Shannon Expansion, and Decision Tree

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th>f</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

\[ f = x' \cdot f_{x=0} + x \cdot f_{x=1} \]

- 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

![Diagram showing the reduction process](Image)
Reduction Rule #2

- Remove redundant tests
  - For a node v, left(v) = right(v)
Reduction Rule #3

- Merge isomorphic nodes
The Reduction Algorithm

- Recursively determine an identifier (id) for each node, starting from leaves; nodes with the same IDs are merged
  - If $id(left(v)) = id(right(v))$, then $id(v) = id(left(v))$
  - If $id(left(v)) = id(left(u))$ and $id(right(v)) = id(right(u))$, then $id(v) = id(u)$
BDDs History

- Proposed by Lee 1959, Akers 1976
  - Idea of representing Boolean function as a rooted DAG with a decision at each vertex

- Popularized by Bryant 1986
  - Further restrictions + efficient algorithms to make a useful data structure (ROBDD)
  - BDD = ROBDD since then
ROBDDs

- **Reduced and Ordered (ROBDD)**
  - 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
More on Variable Ordering

- Follow a total ordering to variables
  - e.g., $x < y < z$
- Variables must appear in the same ascending order along all paths
Canonical Representation

- ROBDD 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 unsatisfiable function
    - “1” unique tautology
Compactness

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

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

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

- 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.
BDD Limitations

- NP-complete 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
  - General scheme: exploit circuit topology
  - Lots of research in ordering algorithms
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? Clock period?

\[ AT_i = \max_{j \in \text{pred}(i)} \{ AT_j \} + d_i \]

Gates are visited in a topological order
STA: Required Times

Assumptions:
- All inputs arrive at time 0
- All gate delays = 1ns (\(d_i = 1\)); all wire delays = 0
- Clock period = 5ns (200MHz)

Question: **Required time** (RT) of each gate output in order to meet the clock period?

\[
RT_i = \min_{j \in \text{succ}(i)} \{ RT_j - d_j \}
\]

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?

\[ \text{Slack}_i = \text{RT}_i - \text{AT}_i \]
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
Before Next Class

- Start early on CORDIC design!
- Next lecture: Front-end compilation and CDFG