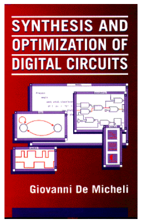
Sequential Logic Synthesis

Giovanni De Micheli

Integrated Systems Laboratory



Module 1

- **Objective**
  - **Motivation and assumptions for sequential synthesis**
  - **Finite-state machine design and optimization**

Synchronous logic circuits

- Interconnection of
  - Combinational logic gates
  - Synchronous delay elements
    - Edge-triggered, master/slave
- Assumptions
  - No direct combinational feedback
  - Single-phase clocking
- Extensions to
  - Multiple-phase clocking
  - Gated latches

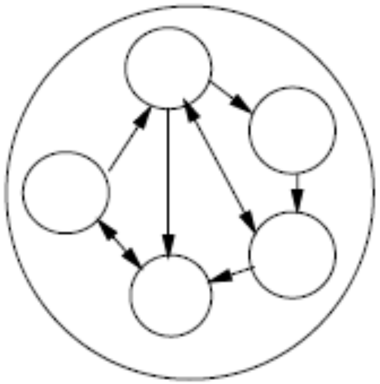
Modeling synchronous circuits

- Circuit are modeled in hardware languages
  - Circuit model may be directly related to FSM model
    - Description by: switch-case
  - Circuit model may be structural
    - Explicit definition of registers
- Sequential circuit models can be generated from high-level models
  - Control generation in high-level synthesis

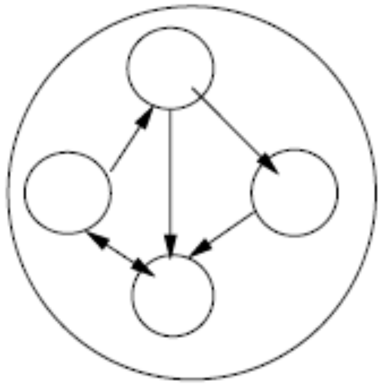
Modeling synchronous circuits

- **State-based** model:
  - Model circuits as finite-state machines (FSMs)
  - Represent by state tables/diagrams
  - Apply exact/heuristic algorithms for:
    - State minimization
    - State encoding
- **Structural** model
  - Represent circuit by synchronous logic network
  - Apply
    - Retiming
    - Logic transformations

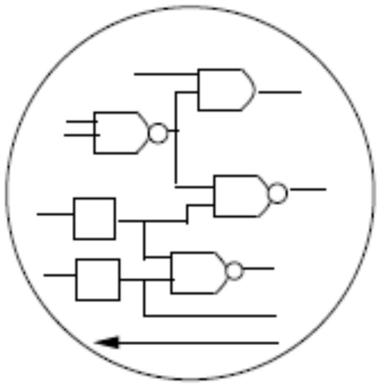
State-based optimization



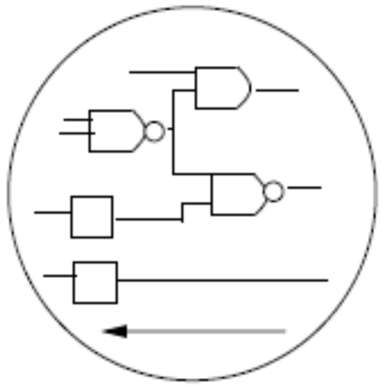
FSM Specification



State Minimization



State Encoding



Combinational Optimization

Modeling synchronous circuits

- Advantages and disadvantages of models
- State-based model
  - Explicit notion of state
  - Implicit notion of area and delay
- Structural model
  - Implicit notion of state
  - Explicit notion of area and delay
- Transition from a model to another is possible
  - State encoding
  - State extraction

Sequential logic optimization

- Typical flow
  - Optimize FSM state model first
    - Reduce complexity of the model
    - E.g., apply state minimization
    - Correlates to area reduction
  - Encode states and obtain a structural model
    - Apply retiming and transformations
    - Achieve performance enhancement
  - Use state extraction for verification purposes

Formal finite-state machine model

- A set of primary input patterns X
- A set of primary output patterns Y
- A set of states S
- A state transition function: $\delta: X \times S \rightarrow S$
- An output function:
  - $\lambda: X \times S \rightarrow Y$ for Mealy models
  - $\lambda: S \rightarrow Y$ for Moore models

State minimization

- **Classic problem**
  - Exact and heuristic algorithms are available
  - Objective is to reduce the number of states and hence the area
- **Completely-specified finite-state machines**
  - No *don't care* conditions
  - Polynomial-time solutions
- **Incompletely-specified finite-state machines**
  - Unspecified transitions and/or outputs
    - Usual case in synthesis
  - Intractable problem:
    - Requires binate covering

State minimization for completely-specified FSMs

- **Equivalent states:**
  - Given any input sequence, the corresponding output sequence match
- **Theorem:**
  - Two states are equivalent if and only if:
    - They lead to identical outputs and their next-states are equivalent
- **Equivalence is transitive**
  - Partition states into equivalence classes
  - Minimum finite-state machine is unique

State minimization for completely-specified FSMs

- **Stepwise partition refinement:**
  - Initially:
    - All states in the same partition block
  - Iteratively:
    - Refine partition blocks
  - At convergence:
    - Partition blocks identify equivalent states
- **Refinement can be done in two directions**
  - Transitions *from* states in block to other states
    - Classic method. Quadratic complexity
  - Transitions *into* states of block under consideration
    - Inverted tables. Hopcroft's algorithm.

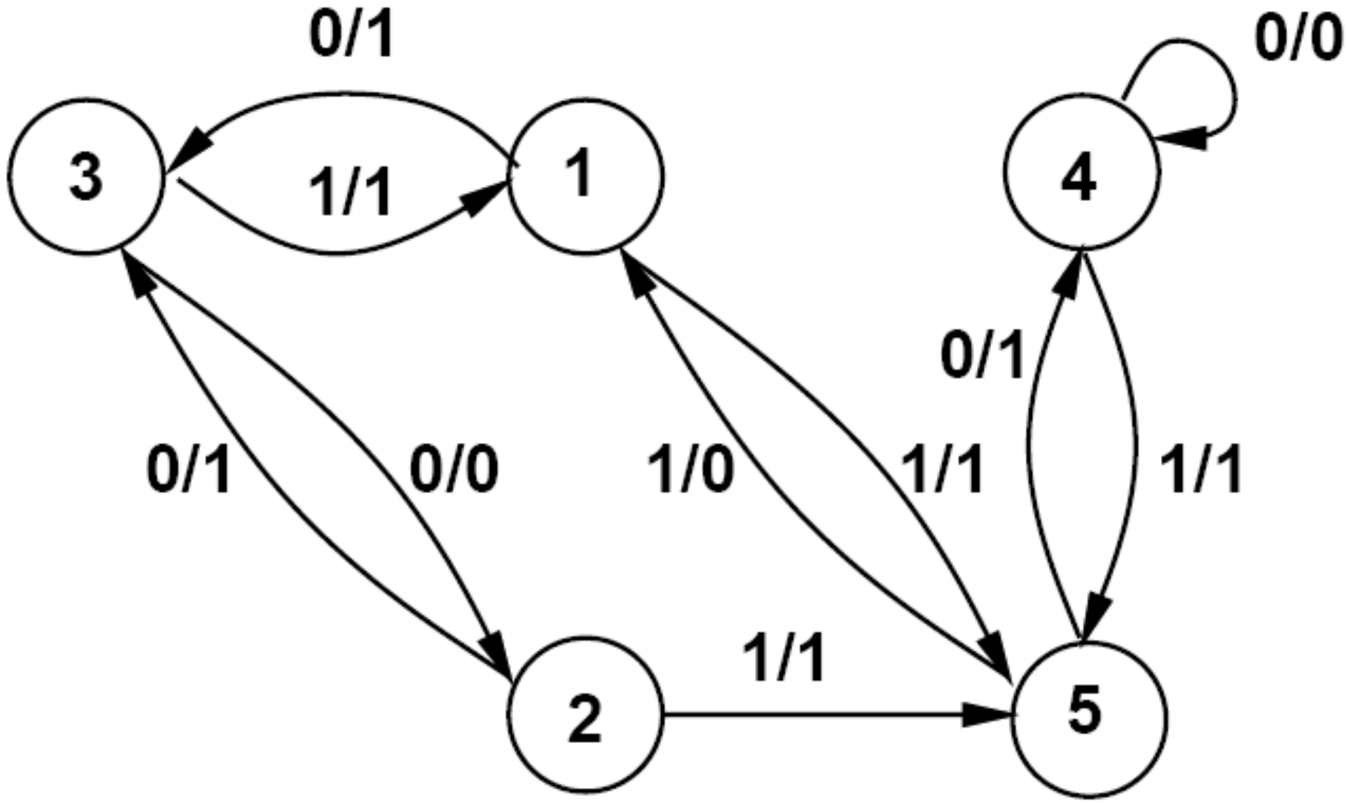
Example of refinement

- **Initial partition:**
  - Π_1 : States belong to the same block when outputs are the same for any input
- **Iteration:**
  - Π_{k+1} : States belong to the same block if they were previously in the same block and their next states are in the same block of Π_k for any input
- **Convergence:**
  - $\Pi_{k+1} = \Pi_k$

Example

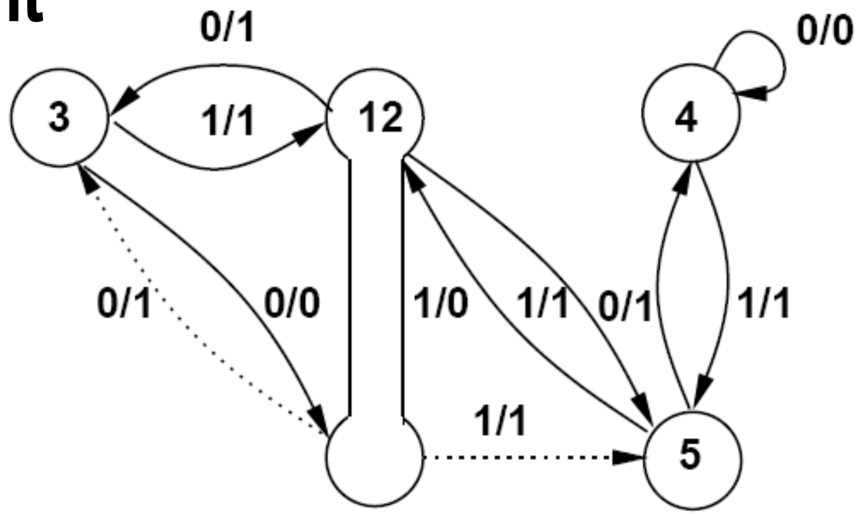
INPUT	STATE	N-STATE	OUTPUT
0	s_1	s_3	1
1	s_1	s_5	1
0	s_2	s_3	1
1	s_2	s_5	1
0	s_3	s_2	0
1	s_3	s_1	1
0	s_4	s_4	0
1	s_4	s_5	1
0	s_5	s_4	1
1	s_5	s_1	0

Example



Example

- $\Pi_1 = \{ \{ s_1, s_2\} , \{ s_3,s_4 \}, \{ s_5 \} \}$
- $\Pi_2 = \{ \{ s_1, s_2\} , \{ s_3\} , \{ s_4 \}, \{ s_5 \} \}$
- Π_2 is a partition into equivalence classes
  - No further refinement is possible
  - States $\{ s_1, s_2 \}$ are equivalent



State minimization for incompletely-specified finite-state machines

- **Applicable input sequences**
  - **All transitions are specified**
- **Compatible states**
  - **Given any applicable input sequence, the corresponding output sequence match**
- **Theorem:**
  - **Two states are compatible if and only if:**
    - **They lead to identical outputs**
      - (when both are specified)
    - **And their next state is compatible**
      - (when both are specified)

State minimization for incompletely-specified finite-state machines

- Compatibility is not an equivalence relation
- Minimum finite-state machine is not unique
- Implication relation make the problem intractable
  - Two states may be compatible, subject to other states being compatible.
  - Implications are binate satisfiability clauses
    - a -> b = a'+b

Example

INPUT	STATE	N-STATE	OUTPUT
0	s_1	s_3	1
1	s_1	s_5	*
0	s_2	s_3	*
1	s_2	s_5	1
0	s_3	s_2	0
1	s_3	s_1	1
0	s_4	s_4	0
1	s_4	s_5	1
0	s_5	s_4	1
1	s_5	s_1	0

Trivial method

- Consider all possible *don't care* assignments
  - n *don't care* imply
    - 2^n completely specified FSMs
    - 2^n solutions
- Example:
  - Replace * by 1
    - $\Pi_1 = \{ \{ s_1, s_2\} , \{ s_3\}, \{s_4\}, \{ s_5\} \}$
  - Replace * by 0
    - $\Pi_1 = \{ \{ s_1, s_5\} , \{ s_2, s_3 , s_4\} \}$

Compatibility and implications Example

- **Compatible states $\{s_1, s_2\}$**
- **If $\{s_3, s_4\}$ are compatible**
  - Then $\{s_1, s_5\}$ are also compatible
- **Incompatible states $\{s_2, s_5\}$**

INPUT	STATE	N-STATE	OUTPUT
0	s_1	s_3	1
1	s_1	s_5	*
0	s_2	s_3	*
1	s_2	s_5	1
0	s_3	s_2	0
1	s_3	s_1	1
0	s_4	s_4	0
1	s_4	s_5	1
0	s_5	s_4	1
1	s_5	s_1	0

Compatibility and implications

- **Compatible pairs:**
  - $\{s_1, s_2\}$
  - $\{s_1, s_5\} \leftarrow \{s_3, s_4\}$
  - $\{s_2, s_4\} \leftarrow \{s_3, s_4\}$
  - $\{s_2, s_3\} \leftarrow \{s_1, s_5\}$
  - $\{s_3, s_4\} \leftarrow \{s_2, s_4\}$ U $\{s_1, s_5\}$
- **Incompatible pairs**
  - $\{s_2, s_5\}$
  - $\{s_3, s_5\}$
  - $\{s_1, s_4\}$
  - $\{s_4, s_5\}$
  - $\{s_1, s_3\}$

INPUT	STATE	N-STATE	OUTPUT
0	s_1	s_3	1
1	s_1	s_5	*
0	s_2	s_3	*
1	s_2	s_5	1
0	s_3	s_2	0
1	s_3	s_1	1
0	s_4	s_4	0
1	s_4	s_5	1
0	s_5	s_4	1
1	s_5	s_1	0

Compatibility and implications

- A class of compatible states is such that all state pairs are compatible
- A class is maximal
  - If not subset of another class
- Closure property
  - A set of classes such that all compatibility implications are satisfied
- The set of maximal compatibility classes
  - Has the closure property
  - May not provide a minimum solution

Maximum compatibility classes

- **Example:**
  - $\{s_1, s_2\}$
  - $\{s_1, s_5\} \leftarrow \{s_3, s_4\}$
  - $\{s_2, s_3, s_4\} \leftarrow \{s_1, s_5\}$
- **Cover with all MCC has cardinality 3**

Exact problem formulation

- **Prime compatibility classes**:
  - Compatibility classes having the property that they are not subset of other classes implying the same (or subset) of classes
- Compute all prime compatibility classes
- Select a minimum number of prime classes
  - Such that all states are covered
  - All implications are satisfied
- Exact solution requires binate cover
- Good approximation methods exists
  - Stamina

Prime compatibility classes

- **Example:**
  - $\{s_1, s_2\}$
  - $\{s_1, s_5\} \leftarrow \{s_3, s_4\}$
  - $\{s_2, s_3, s_4\} \leftarrow \{s_1, s_5\}$
- **Minimum cover:**
  - $\{s_1, s_5\}$, $\{s_2, s_3, s_4\}$
  - Minimun cover has cardinality 2

State encoding

- Determine a binary encoding of the states
  - Optimizing some property of the representation (mainly area)
- Two-level model for combinational logic
  - Methods based on symbolic optimization
    - Minimize a symbolic cover of the finite state machine
    - Formulate and solve a constrained encoding problem
- Multiple-level model
  - Some heuristic methods that look for encoding which privilege cube and/or kernel extraction
  - Weak correlation with area minimality

Example

INPUT	P-STATE	N-STATE	OUTPUT
0	s1	s3	0
1	s1	s3	0
0	s2	s3	0
1	s2	s1	1
0	s3	s5	0
1	s3	s4	1
0	s4	s2	1
1	s4	s3	0
0	s5	s2	1
1	s5	s5	0

Example

- **Minimum symbolic cover:**

*	s1s2s4	s3	0
1	s2	s1	1
0	s4s5	s2	1
1	s3	s4	1

- **Encoded cover :**

*	1**	001	0
1	101	111	1
0	*00	101	1
1	001	100	1

Summary
finite-state machine optimization

- FSM optimization has been widely researched
  - Classic and newer approaches
- State minimization and encoding correlate to area reduction
  - Useful, but with limited impact
- Performance-oriented FSM optimization has mixed results
  - Performance optimization is usually done by structural methods

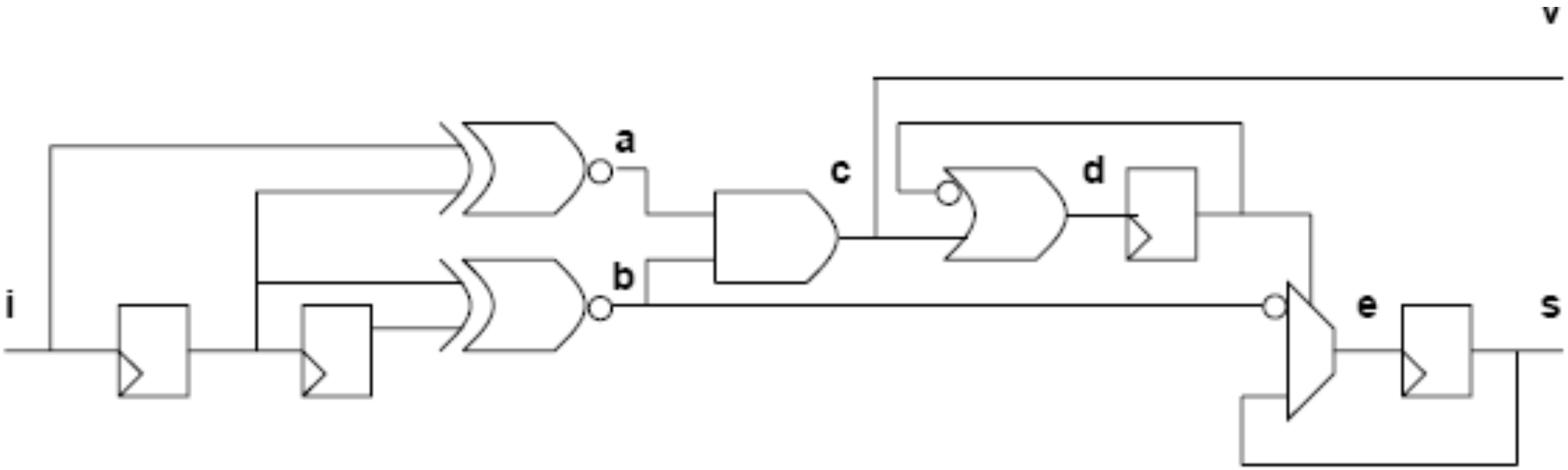
Module 2

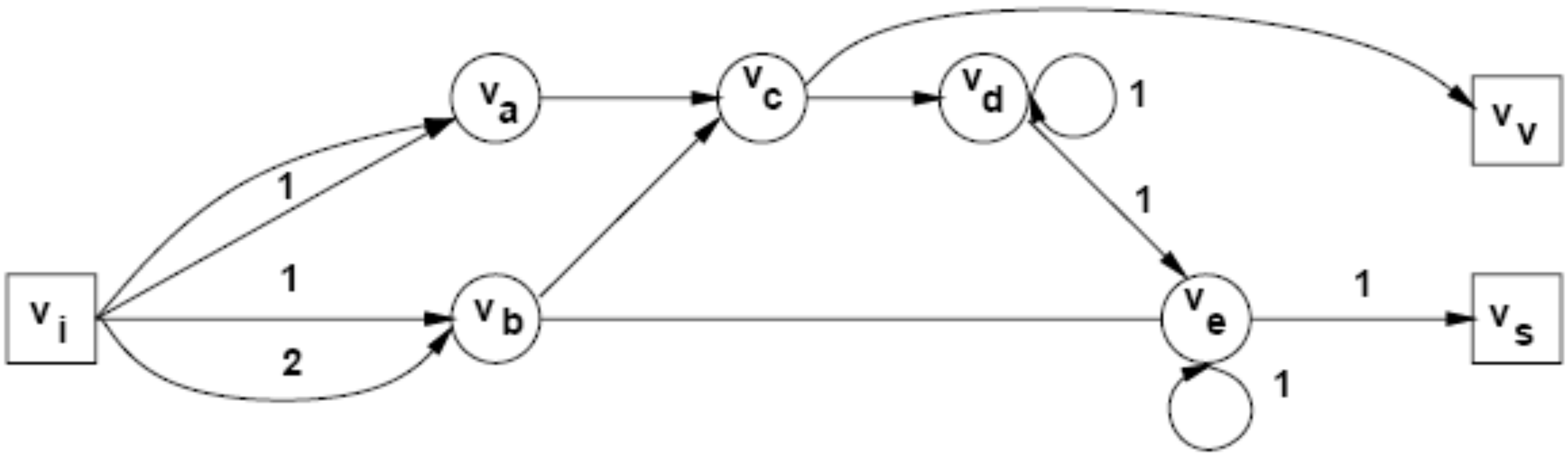
- **Objective**
  - **Structural representation of sequential circuits**
  - **Retiming**
  - **Extensions**

Structural model for sequential circuits

- Synchronous logic network
  - Variables
  - Boolean equations
  - Synchronous delay annotation
- Synchronous network graph
  - Vertices ↔ equations ↔ I/O, gates
  - Edges ↔ dependencies ↔ nets
  - Weights ↔ synchronous delays ↔ registers

Example





Example

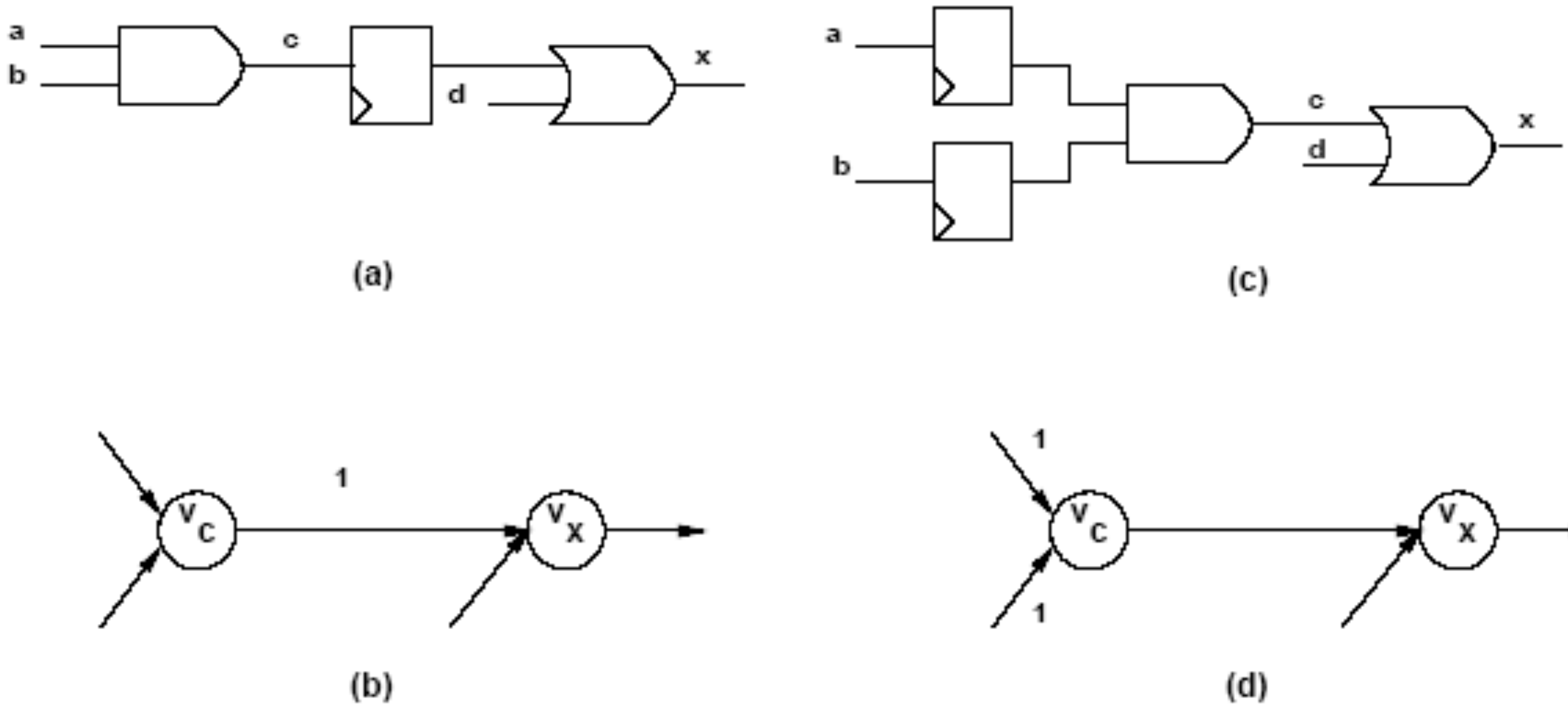
$$
\begin{aligned}
a^{(n)} &= i^{(n)} \,\overline{\oplus}\, i^{(n-1)} \\
b^{(n)} &= i^{(n-1)} \,\overline{\oplus}\, i^{(n-2)} \\
c^{(n)} &= a^{(n)} b^{(n)} \\
d^{(n)} &= c^{(n)} + d'^{(n-1)} \\
e^{(n)} &= d^{(n)} e^{(n-1)} + d'^{(n)} b'^{(n)} \\
v^{(n)} &= c^{(n)} \\
s^{(n)} &= e^{(n-1)}
\end{aligned}
$$

$$
\begin{aligned}
a &= i \,\overline{\oplus}\, i@1 \\
b &= i@1 \,\overline{\oplus}\, i@2 \\
c &= a \; b \\
d &= c + d@1' \\
e &= d \; e@1 + d' \; b' \\
v &= c \\
s &= e@1
\end{aligned}
$$

Approaches to sequential synthesis

- Optimize combinational logic only
  - Freeze circuit at register boundary
  - Modify equation and network graph topology
- Retiming
  - Move register positions. Change weights on graph
  - Preserve network topology
- Synchronous transformations
  - Blend combinational transformations and retiming
  - Powerful, but complex to use

Example of local retiming



Retiming

- Global optimization technique
- Change register positions
  - Affects area:
    - Retiming changes register count
  - Affects cycle-time:
    - Changes path delays between register pairs
- Retiming algorithms have polynomial-time complexity

Retiming assumptions

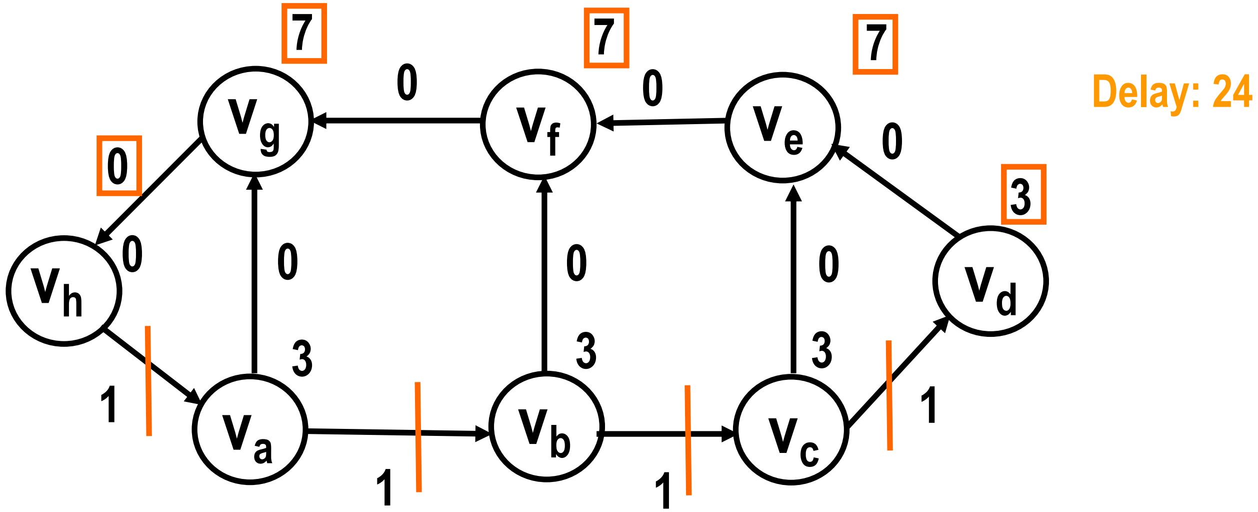
- Delay is constant at each vertex
  - No fanout delay dependency
- Graph topology is invariant
  - No logic transformations
- Synchronous implementation
  - Cycles have positive weights
    - Each feedback loop has to be broken by at least one register
  - Edges have non-negative weights
    - Physical registers cannot anticipate time
- Consider topological paths
  - No false path analysis

Retiming

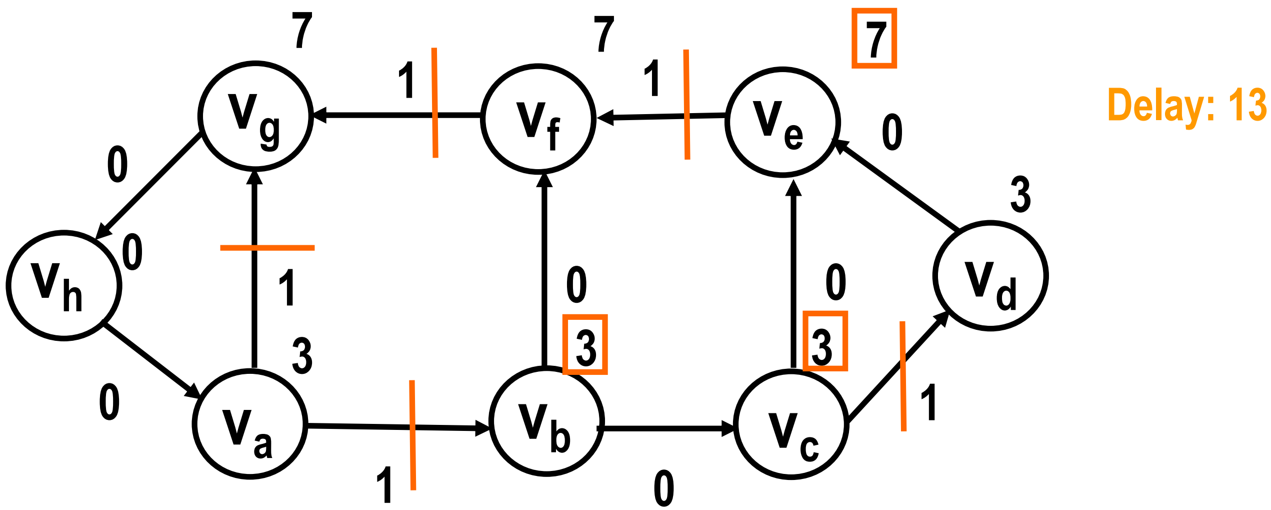
- Retiming of a vertex v
  - Integer r_v
  - Registers moved from output to input: r_v positive
  - Registers moved from input to output: r_v negative
- Retiming of a network
  - Vector whose entries are the retiming at various vertices
- A family of I/O equivalent networks are specified by:
  - The original network
  - A set of vectors satisfying specific constraints
    - Legal retiming

Example

Original graph



Retimed graph



Definitions and properties

- **Definitions:**
  - $w(v_i, v_j)$ weight on edge (v_i, v_j)
  - $(v_i, \ldots, v_j)$ path from v_i to v_j
  - $w(v_i, \ldots, v_j)$ weight on path from v_i to v_j
  - $d(v_i, \ldots, v_j)$ combinational delay on path from v_i to v_j
- **Properties:**
  - Retiming of an edge (v_i, v_j)
    - $\hat{w}_{ij} = w_{ij} + r_j - r_i$

  - Retiming of a path $(v_i, \ldots, v_j)$
    - $\hat{w}(v_i, \ldots, v_j) = w(v_i, \ldots, v_j) + r_j - r_i$
  - Cycle weights are invariant

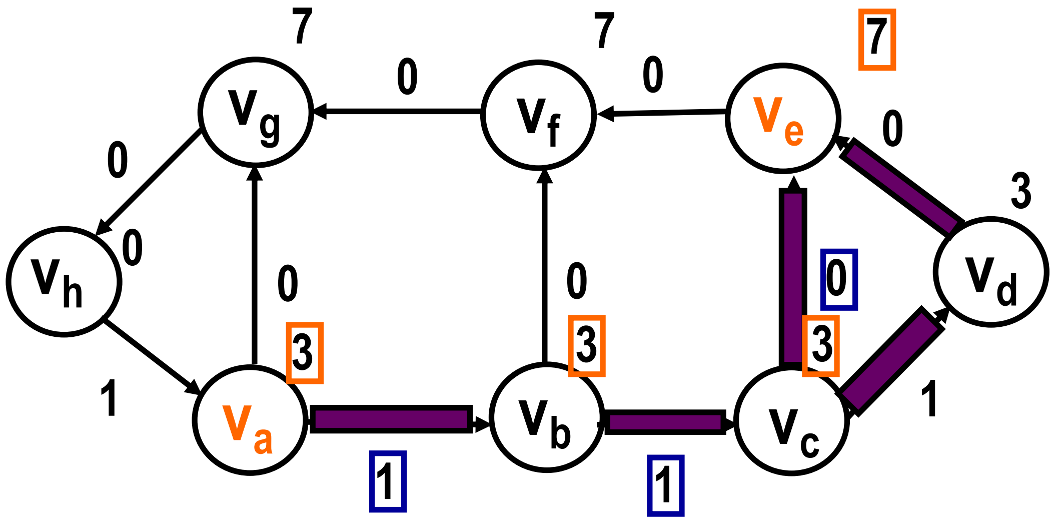
Legal retiming

- **A retiming vector is legal iff:**
  - No edge weight is negative
    - $\hat{w}_{ij}(v_i, v_j) = w_{ij}(v_i, v_j) + r_j - r_i \geq 0$ for all i, j
  - Given a clock period φ:
  - Each path $(v_i, \ldots, v_j)$ with $d(v_i, \ldots, v_j) > \varphi$ has at least one register:
    - $\hat{w}(v_i, \ldots, v_j) = w(v_i, \ldots, v_j) + r_j - r_i \geq 1$ for all i, j
  - Equivalently, each combinational path delay is less than φ

Refined analysis

- **Least-register path**
  - $W(v_i, v_j) = \min w(v_i, \ldots, v_j)$ over all paths between v_i and v_j
- **Critical delay:**
  - $D(v_i, v_j) = \max d(v_i, \ldots, v_j)$ over all paths between v_i and v_j with weight $W(v_i, v_j)$
- **There exist a vertex pair (v_i, v_j) whose delay $D(v_i, v_j)$ bounds the cycle time**

Example



- Vertices: v_a, v_e
- Paths: (v_a, v_b, v_c , v_e) and (v_a, v_b, v_c, v_d, v_e)
- $W(v_a, v_e) = 2$
- $D(v_a, v_e) = 16$

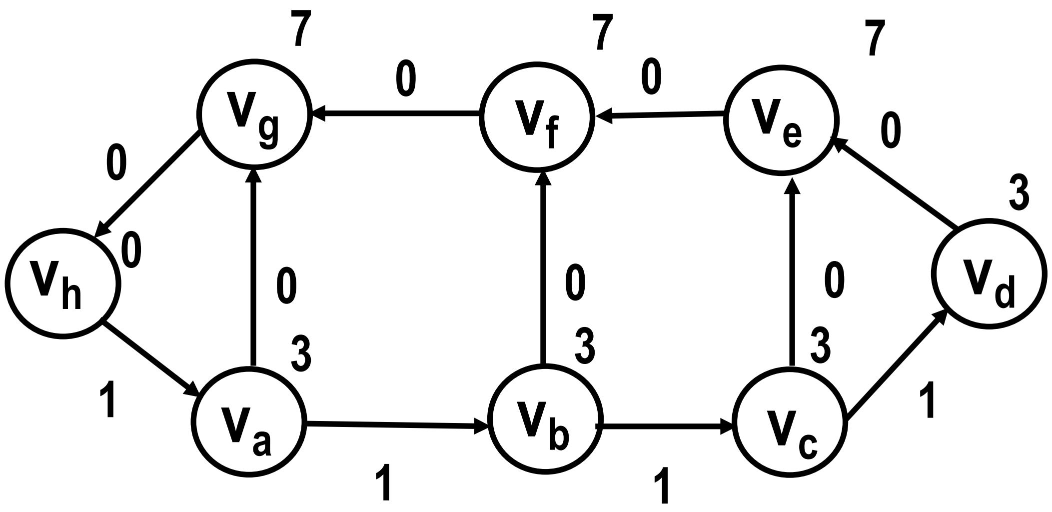
Minimum cycle-time retiming problem

- Find the minimum value of the clock period φ such that there exist a retiming vector where:
  - $r_i - r_j \leq w_{ij}$ for all (v_i, v_j)
    - All registers are implementable
  - $r_i - r_j \leq W(v_i, v_j) - 1$ for all (v_i, v_j) such that $D(v_i, v_j) > \varphi$
    - All timing path constraints are satisfied
- Solution
  - Given a value of φ
  - Solve linear constraints $A\,r \leq b$
    - Mixed integer-linear program
  - A set of inequalities has a solution if the constraint graph has no positive cycles
    - Bellman-Ford algorithm – compute longest path
  - Iterative algorithm
    - Relaxation

Minimum cycle-time retiming algorithm

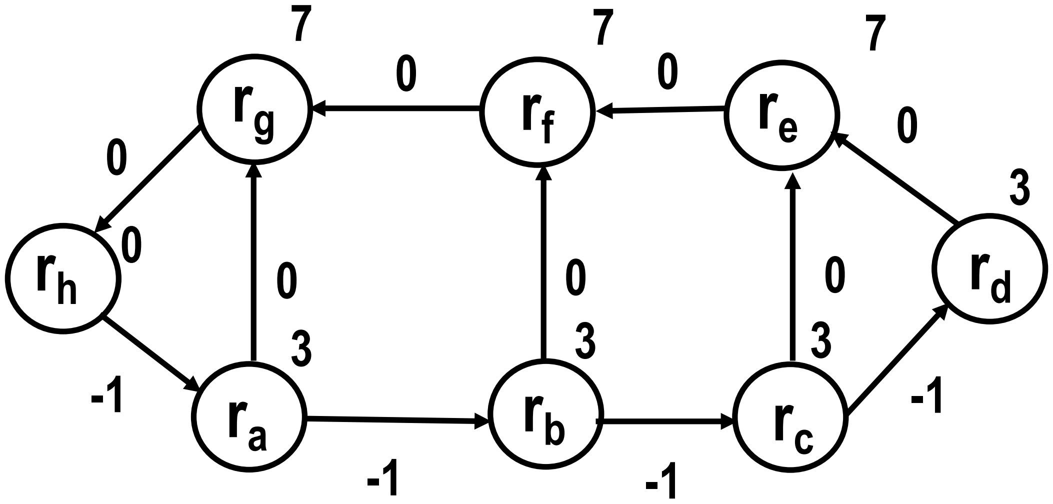
- Compute *all pair* path weights $W(v_i, v_j)$ and delays $D(v_i, v_j)$
  - Warshall-Floyd algorithm with complexity $O(|V|^3)$
- Sort the elements of $D(v_i, v_j)$ in decreasing order
  - Because an element of D is the minimum φ
- Binary search for a φ in $D(v_i, v_j)$ such that
  - There exists a legal retiming
  - Bellman-Ford algorithm with complexity $O(|V|^3)$
- Remarks
  - Result is a global optimum
  - Overall complexity is $O(|V|^3 \log |V|)$

Example: original graph

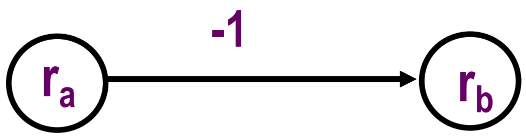


- Constraints (first type):
  - $r_a - r_b \leq 1$ or equivalently $r_b \geq r_a - 1$
  - $r_c - r_b \leq 1$ or equivalently $r_c \geq r_b - 1$
  - …

Example: constraint graph



- **Constraints (first type):**
  - **$r_a - r_b \leq 1$ or equivalently $r_b \geq r_a - 1$**
  - **$r_c - r_b \leq 1$ or equivalently $r_c \geq r_b - 1$**
  - **…**



Example

- Sort elements of D:
  - 33,30,27,26,24,23,21,20,19,17,16,14,13,12,10,9,7,6,3
- Select $\varphi = 19$
  - 33,30,27,26,24,23,21,20,19,17,16,14,13,12,10,9,7,6,3
  - Pass: legal retiming found
- Select $\varphi = 13$
  - 33,30,27,26,24,23,21,20,19,17,16,14,13,12,10,9,7,6,3
  - Pass: legal retiming found
- Select $\varphi < 13$
  - 33,30,27,26,24,23,21,20,19,17,16,14,13,12,10,9,7,6,3
  - Fail: no legal retiming found
- Fastest cycle time is $\varphi = 13$. Corresponding retiming vector is used

Example $\varphi = 13$

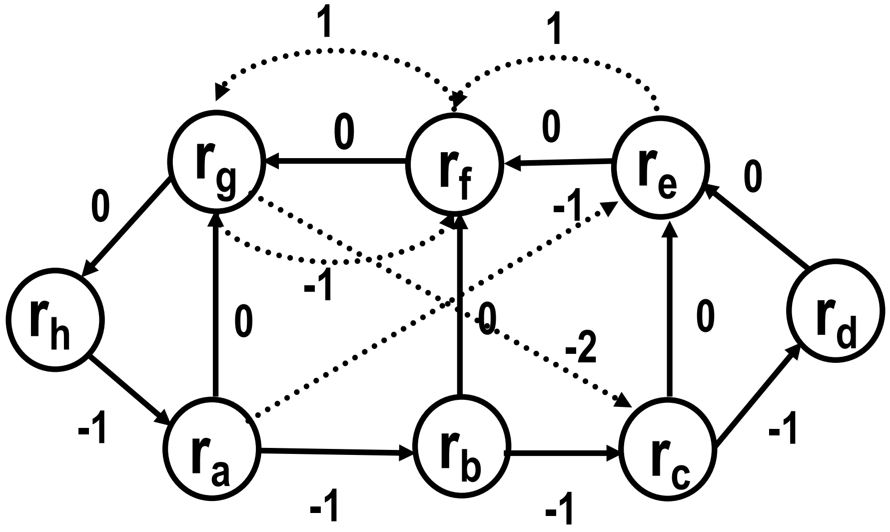
$r_a - r_e \leq 2-1$ or equivalently $r_e \geq r_a - 1$

$r_e - r_f \leq 0-1$ or equivalently $r_f \geq r_e + 1$

$r_f - r_g \leq 0-1$ or equivalently $r_g \geq r_f + 1$

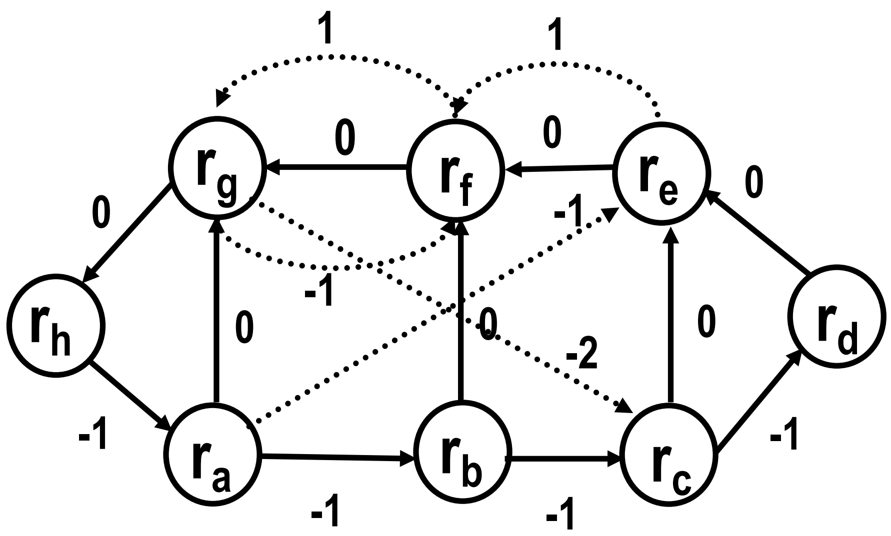
$r_g - r_f \leq 2-1$ or equivalently $r_f \geq r_g - 1$

$r_g - r_c \leq 3-1$ or equivalently $r_c \geq r_g - 2$



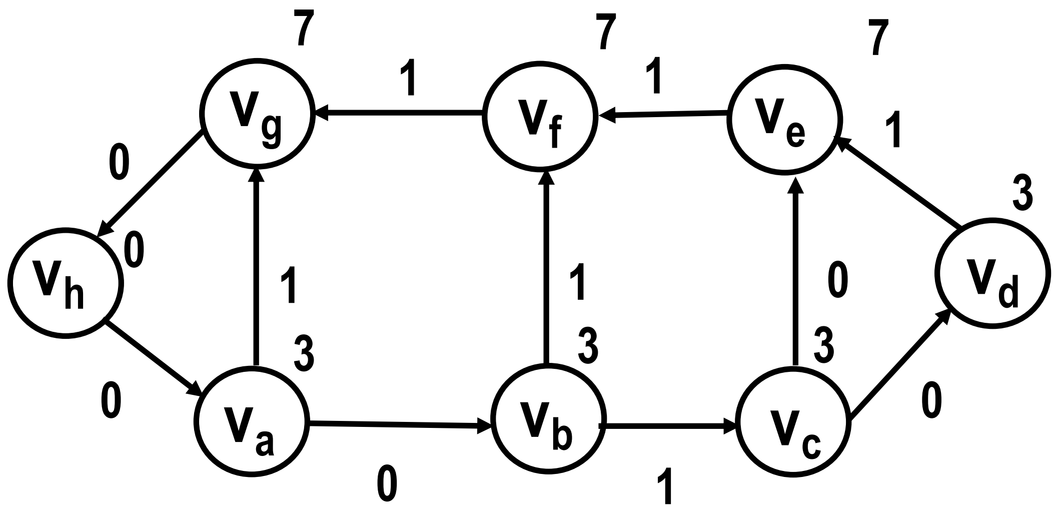
Example φ = 13

- **Constraint graph:**



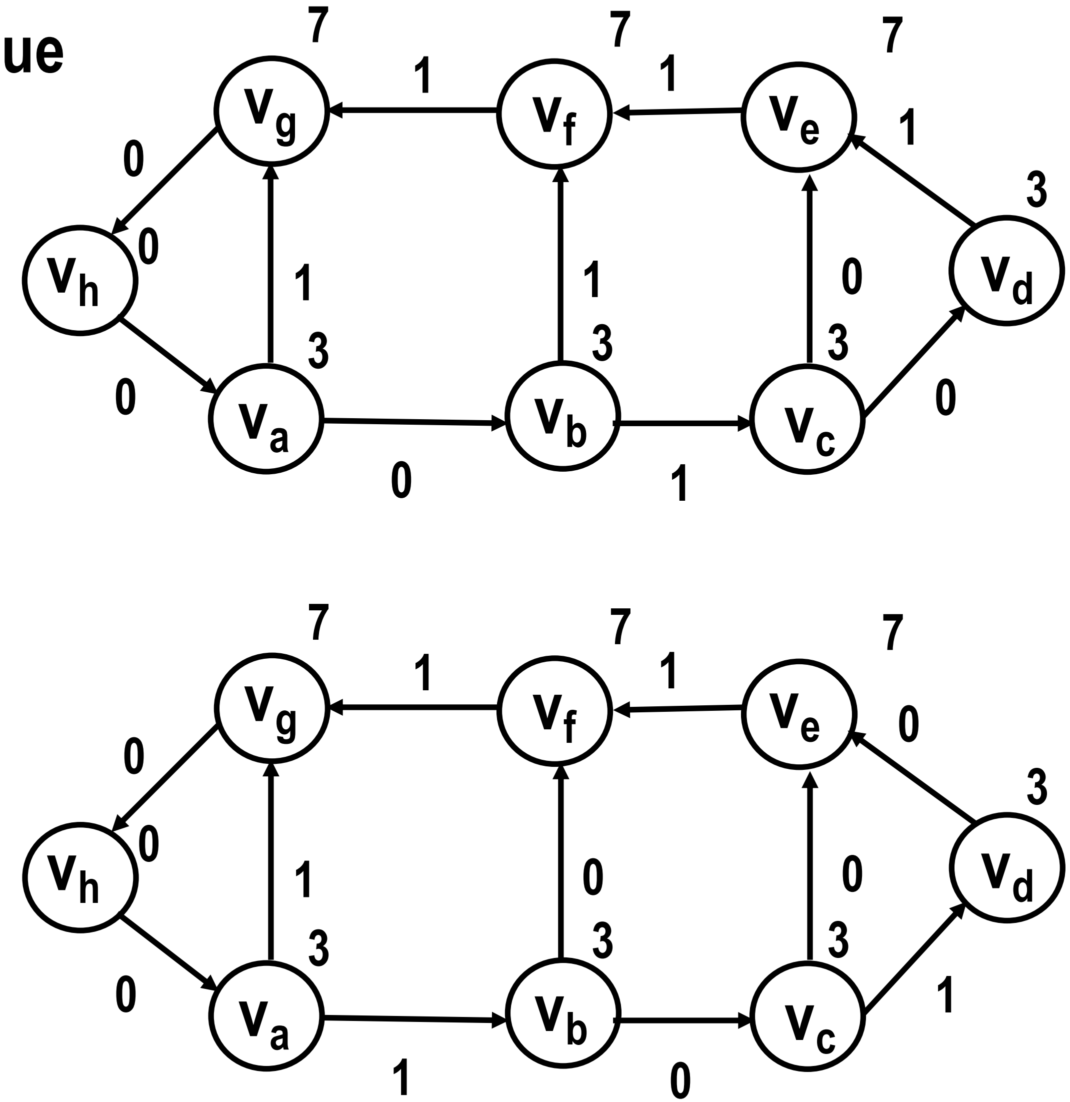
- **Longest path from source**
  - -[12232100]

- **Retimed graph**



Example φ = 13

- The solution is not unique



Relaxation-based retiming

- Most common algorithm for retiming
  - Avoids storage of matrices **W** and **D**
  - Applicable to large circuits
- Rationale
  - Search for decreasing φ in fixed step
    - Look for values of φ compatible with peripheral circuits
  - Use efficient method to determine legality
    - Network graph is often very sparse
  - Can be coupled with topological timing analysis

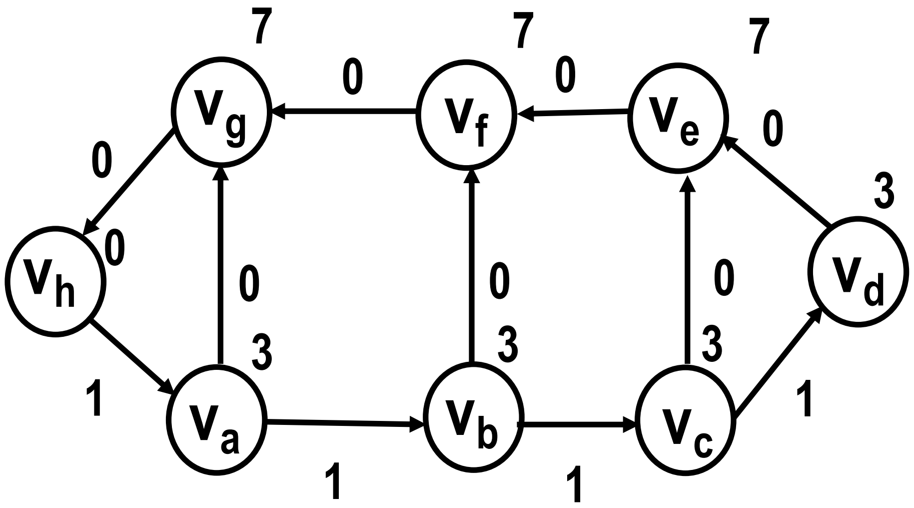
Relaxation-based retiming

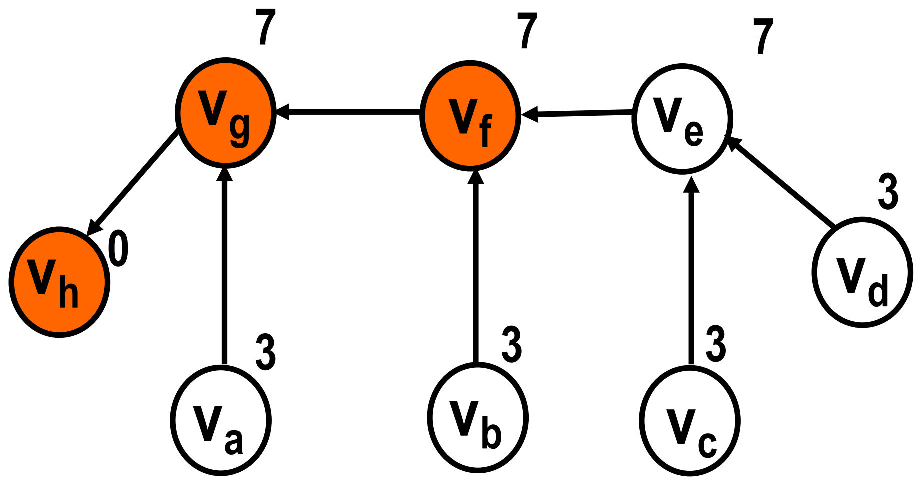
- Start with a given cycle-time φ
- Look for paths with excessive delays
- Make such paths shorter
  - By bringing the terminal register closer
  - Some other paths may become longer
  - Namely, those path whose tail has been moved
- Use an iterative approach

Relaxation-based retiming

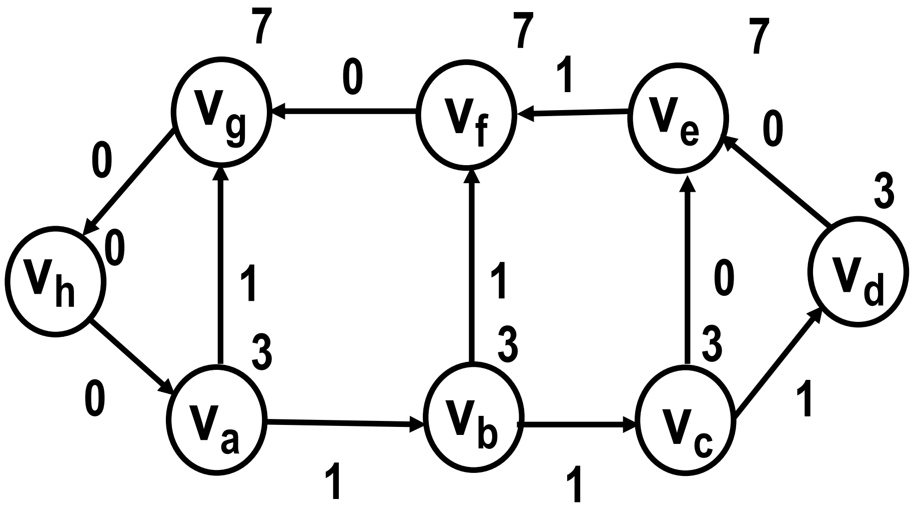
- **Define data ready time at each node**
  - Total delay from register boundary
- **Iterative approach**
  - Find vertices with *data ready* > φ
  - Retime these vertices by 1
- **Algorithm properties**
  - If at some iteration there is no vertex with *data ready* > φ, a legal retiming has been found
  - If a legal retiming is not found in $|V|$ iterations, then no legal retiming exists for that φ

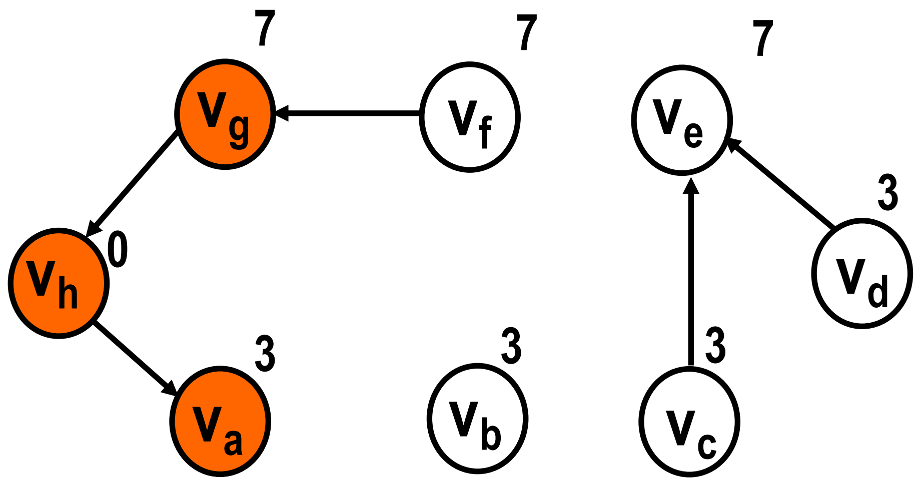
Example $\varphi = 13$ iteration = 1



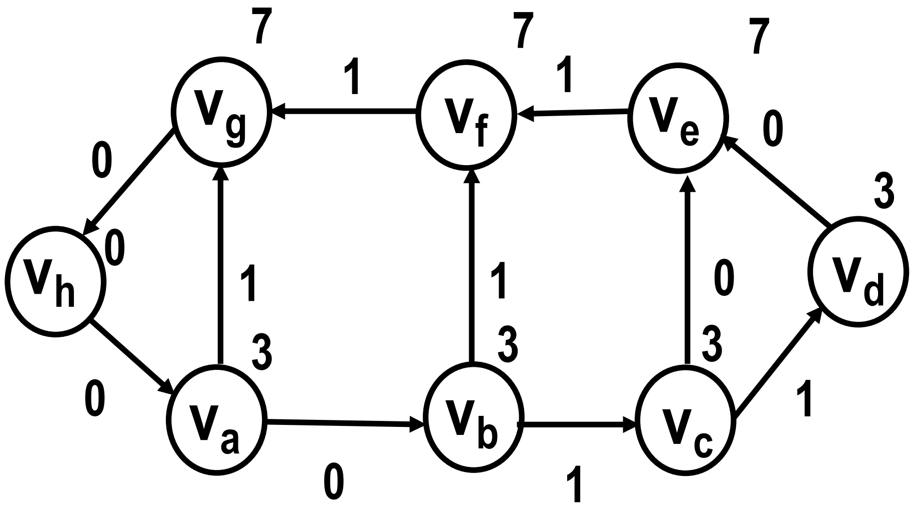


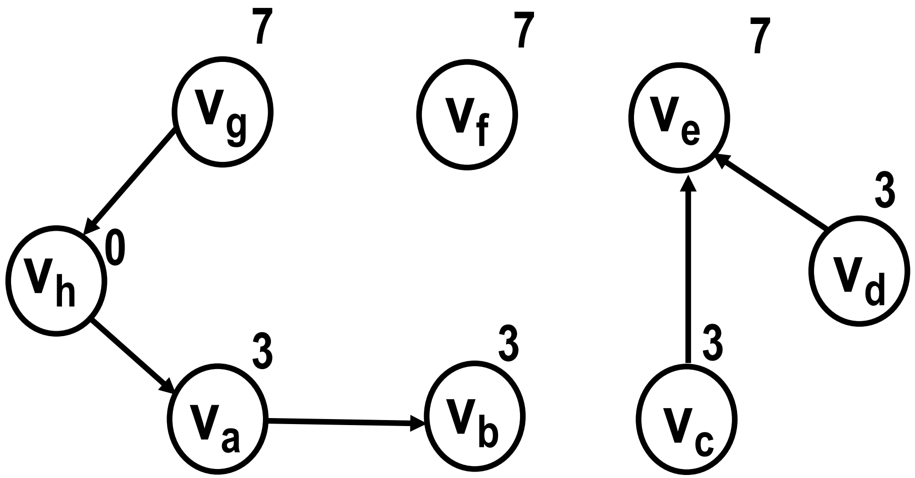
Example $\varphi = 13$ iteration = 2





Example $\varphi = 13$ iteration = 3

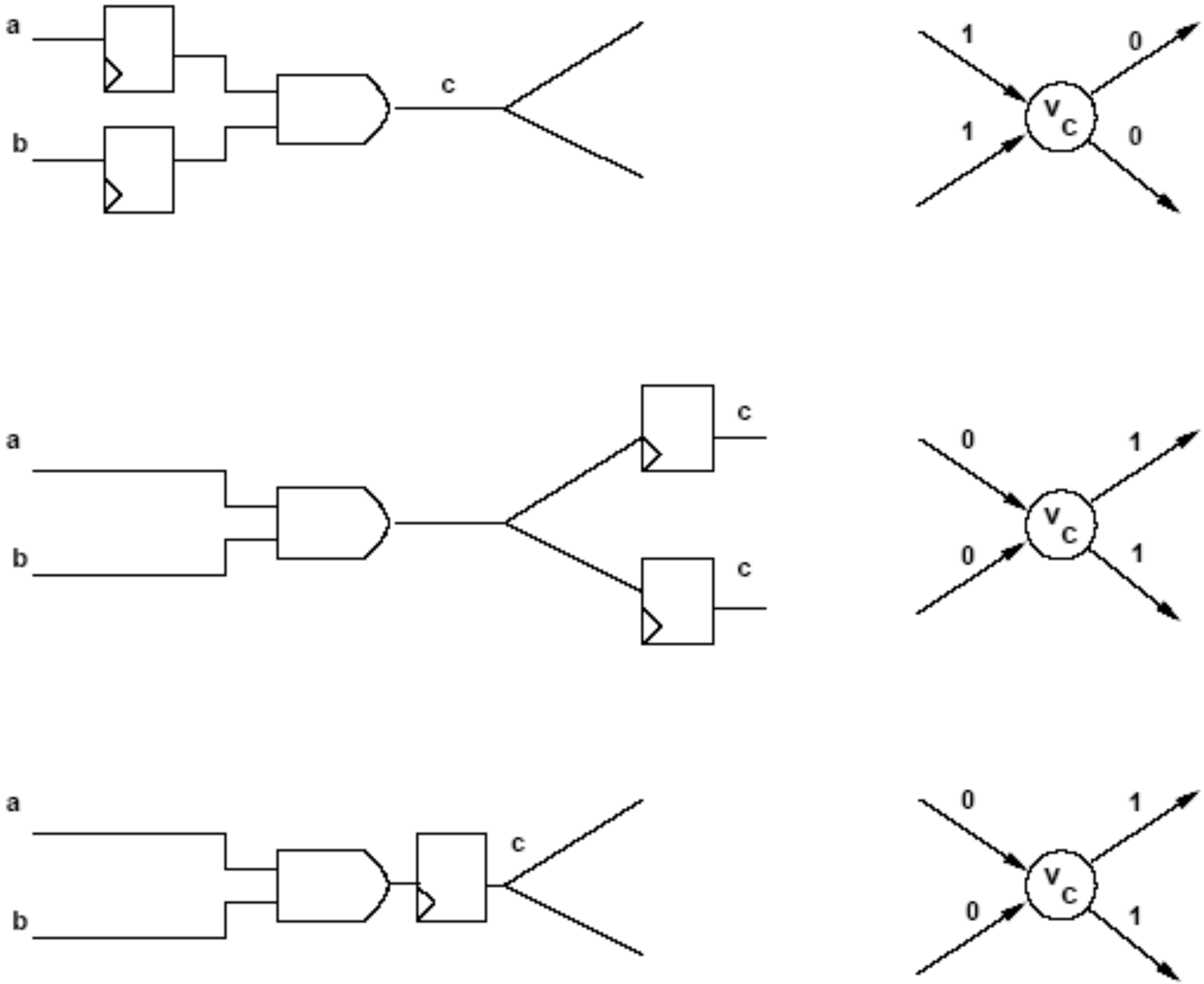




Retiming for minimum area

- Find a retiming vector that minimizes the number of registers
- Simple area modeling
  - Every edge with a positive weight denotes registers
  - Total register area is proportional to the sum of all weights
- Register sharing model
  - Every set of positively-weighted edges with common tail is realized by a shift register with taps
  - Total register area is proportional to the sum, over all vertices, of the maxima of weights on outgoing edges

Example



Minimum area retiming simple model

- Register variation at node v
  - r_v (indegree(v) – outdegree(v))
- Total area variation:
  - Σ r_v (indegree(v) – outdegree(v))
- Area minimization problem:
  - Min Σ r_v (indegree(v) – outdegree(v))
  - Such that $r_i - r_j \leq w_{ij}$ for all (v_i, v_j)

Minimum area retiming under timing constraint

- Area recovery under timing constraint
  - Min $\Sigma\, r_v$ (indegree(v) – outdegree(v)) such that:
  - $r_i - r_j \le w_{ij}$ for all (v_i, v_j) and
  - $r_i - r_j \le W(v_i, v_j) - 1$ for all (v_i, v_j) such that $D(v_i, v_j) > \varphi$
- Common implementation is by integer linear program
  - Problem can alternatively be transformed into a matching problem and solved by Edmonds-Karp algorithm
- Register sharing
  - Construct auxiliary network and apply this formulation.
  - Auxiliary network construction takes into account register sharing

Other problems related to retiming

- Retiming pipelined circuits
  - Balance pipe stages by using retiming
  - Trade-off latency versus cycle time
- Peripheral retiming
  - Use retiming to move registers to periphery of a circuit
  - Restore registers after optimizing combinational logic
- Wire pipelining
  - Use retiming to pipeline interconnection wires
  - Model sequential and combinational macros
  - Consider wire delay and buffering

Summary of retiming

- Sequential optimization technique for:
  - Cycle time or register area
- Applicable to
  - Synchronous logic networks
  - Architectural models of data paths
    - Vertices represent complex (arithmetic) operators
  - Exact algorithm in polynomial time
- Extension and issues
  - Delay modeling
  - Network granularity

Module 3

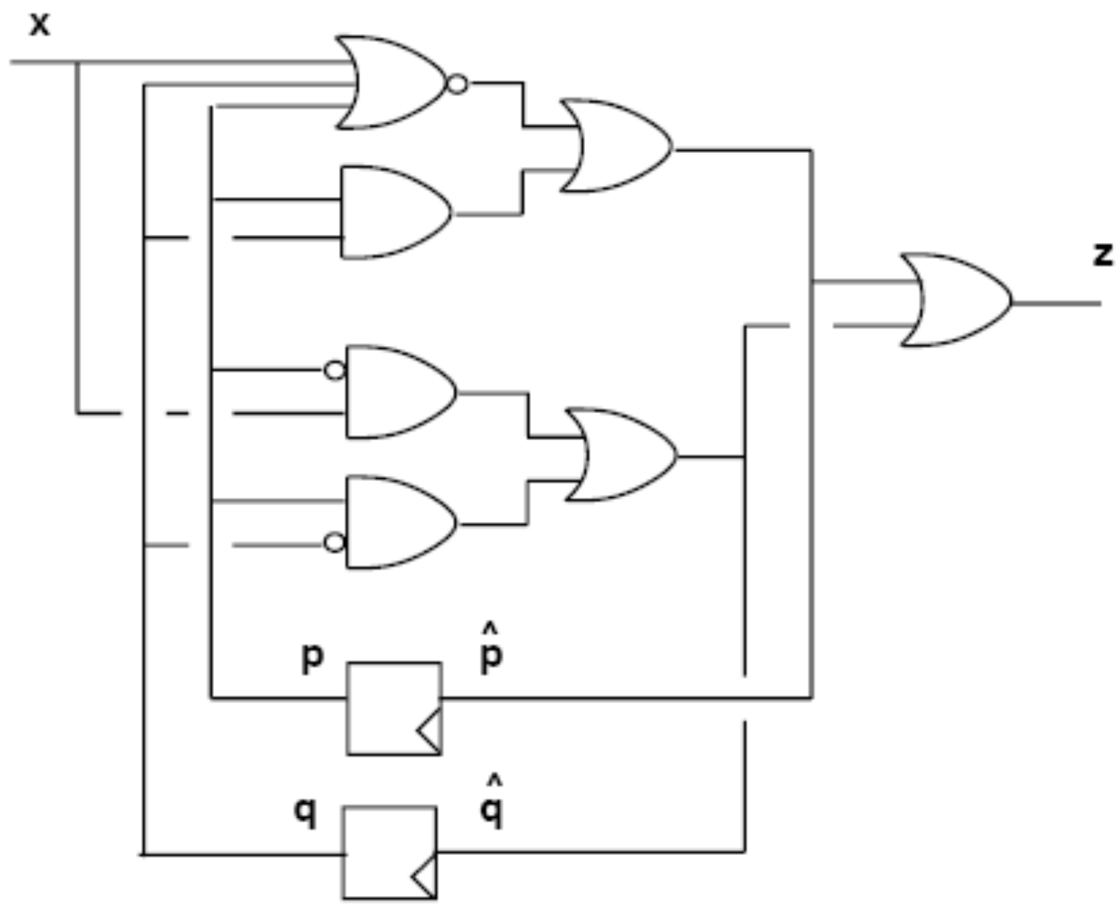
- **Objective**
  - **Relating state-based and structural models**
  - **State extraction**

Relating the sequential models

- **State encoding**
  - **Maps a state-based representation into a structural one**
- **State extraction**
  - **Recovers the state information from a structural model**
- **Remark**
  - **A circuit with n registers may have 2^n states**
  - **Unreachable states**

State extraction

- State variables: p, q
- Initial state p=0; q=0;
- Four possible states



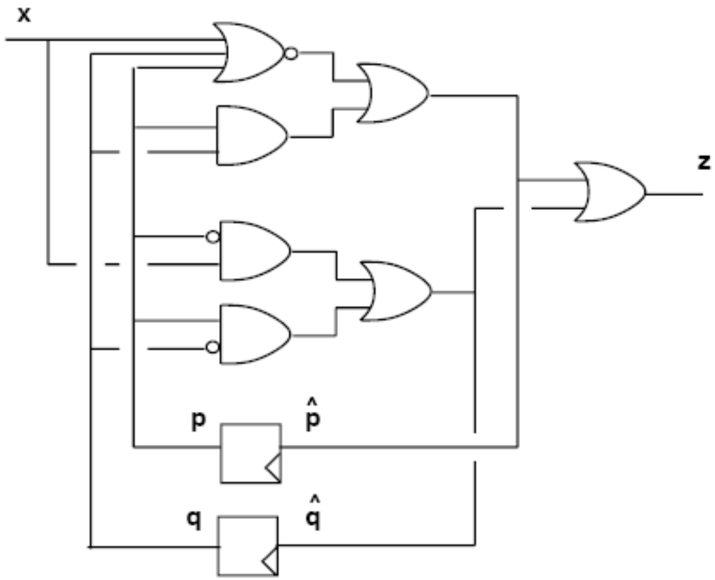
State extraction

- **Reachability analysis**
  - Given a state, determine which states are reachable for some inputs
  - Given a state subset, determine the reachable state subset
  - Start from an initial state
  - Stop when convergence is reached
- **Notation:**
  - A state (or a state subset) is represented by an expression over the state variables
  - Implicit representation

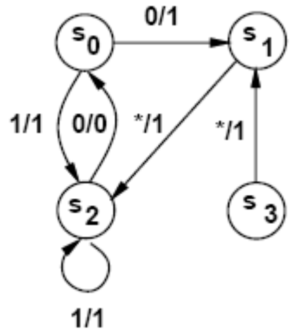
Reachability analysis

- State transition function: f
- Initial state: r_0
- States reachable from r_0
  - Image of r_0 under f
- States reachable from set r_k
  - Image of r_k under f
- Iteration
  - $r_{k+1} = r_k \cup$ (image of r_k under f)
- Convergence
  - $r_{k+1} = r_k$ for some k

Example



(a)



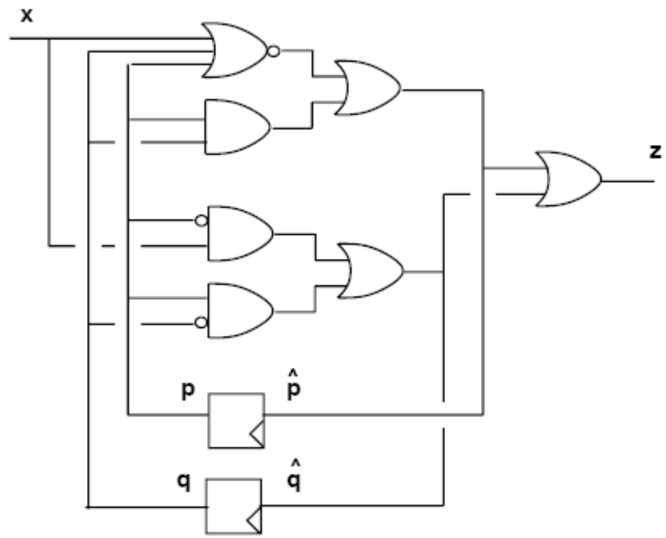
(b)

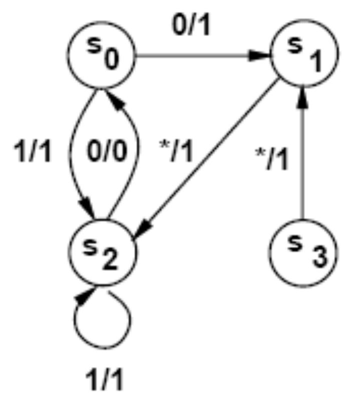
- Initial state $r_0 = p'q'$.

- The state transition function $\mathbf{f} = \begin{bmatrix} x'p'q' + pq \\ xp' + pq' \end{bmatrix}$

Example

- Image of $p'q'$ under f:
  - When ($p = 0$ and $q = 0$), f reduces to $[\, x' \;\; x \,]^T$
  - Image is $[\,0\ 1\,]^T \cup [\,1\ 0\,]^T$
- States reachable from the reset state:
  - ($p = 1; q = 0$) and ($p = 0; q = 1$)
  - $r_1 = p'q' + pq' + p'q = p' + q'$
- States reacheable from r_1:
  - $[\,0\ 0\,]^T \cup [\,0\ 1\,]^T \cup [\,1\ 0\,]^T$
- Convergence:
  - $s_0 = p'q'$; $s_1 = pq'$; $s_2 = p'q$;





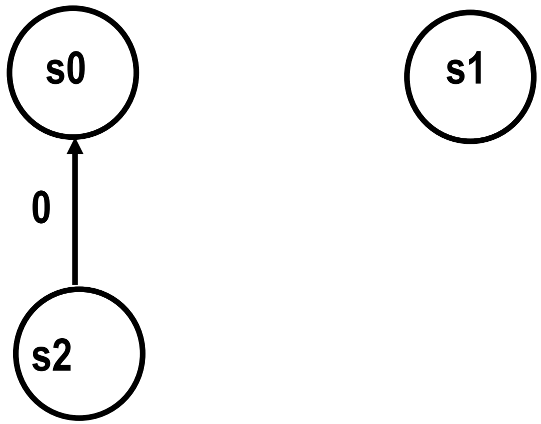
Completing the extraction

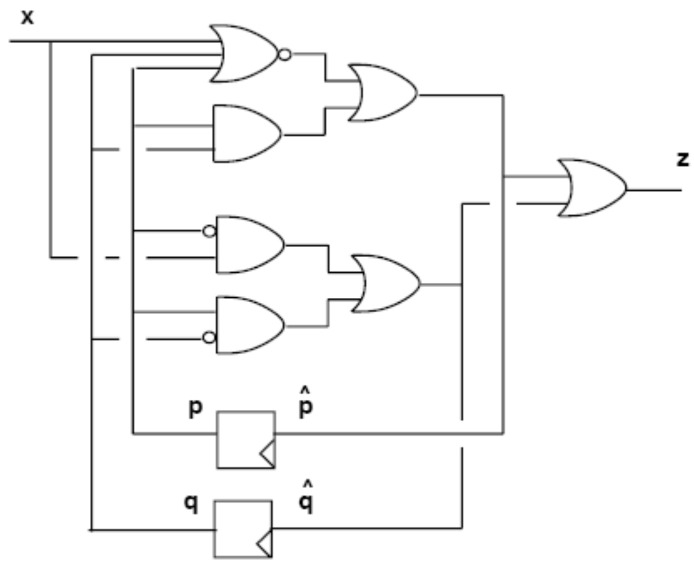
- Determine state set
  - Vertex set
- Determine transitions and I/O labels
  - Edge set
  - Inverse image computation
  - Look at conditions that lead into a given state

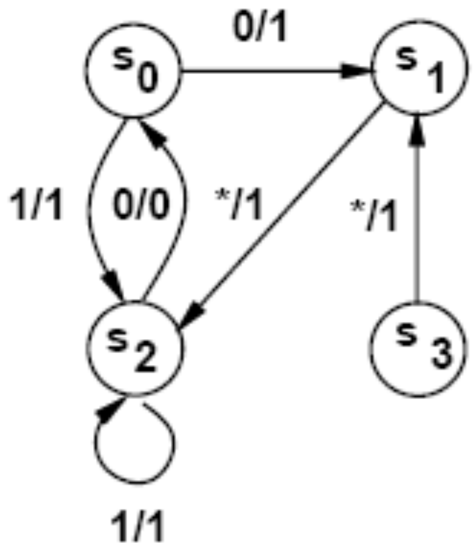
Example

- Transition into $s_0 = p' \, q'$
  - Patterns that make $f = [\, 0 \ 0 \,]^T$ are:

    $(x' \, p' \, q' \; + pq)' \, (xp' \; + pq' \,)' \; = x' \, p' \, q$
  - Transition from state $s_2 = p' \, q$ under input x'
  - And so on …







Remarks

- Extraction is performed efficiently with implicit methods
- Model transition relation χ (i,x,y) with BDDs
  - This function relates possible triples:
    - (input, current_state, next_state)
  - Image of r_k:
    - $S_{i,x}$ (χ(i,x,y) r_k (x))
    - Where r_k depends on inputs x
  - Smoothing on BDDs can be achieved efficiently

Summary

- State extraction can be performed efficiently to:
  - Apply state-based optimization techniques
  - Apply verification techniques
- State extraction is based on forward and backward state space traversal:
  - Represent state space implicitly with BDDs
  - Important to manage the space size, which grows exponentially with the number of registers