Combinational & Sequential Circuit Design

Dr. Mahdi Abbasi
Bu Ali Sina University
Department of Computer Engineering

Outline

- Definitions
- Boolean Expansion Based on Orthonormal Basis
- Sum of Product (SOP) Simplification Procedure
- SOP Simplification Procedure using Don’t Cares
- Iterative Arithmetic Combinational Circuits
- Sequential Circuit Model
- Sequential Circuit Design Procedure
- State Minimization
- State Encoding
- Retiming
- Sequential Circuit Timing

Definitions

- A product term of a function is said to be an implicant.
- A Prime Implicant (PI) is a product term obtained by combining the maximum possible number of adjacent 1-squares in the map.
- A Prime Implicant is a product that we cannot remove any of its literals.
- If a minterm is covered only by one prime implicant then this prime implicant is said to be an Essential Prime Implicant (EPI).

Definitions

- A cover of a Boolean function is a set of implicants that covers its minterms.
- Minimum cover
  - Cover of the function with minimum number of implicants.
  - Global optimum.
- Minimal cover or irredundant cover
  - Cover of the function that is not a proper superset of another cover.
  - No implicant can be dropped.
  - Local optimum.

Definitions

- Let $f(x_1,x_2,\ldots,x_n)$ be a Boolean function of n variables.
- The *cofactor* of $f(x_1,x_2,\ldots,x_i,\ldots,x_n)$ with respect to variable x_i is $f_{xi} = f(x_1,x_2,\ldots,x_i=1,\ldots,x_n)$
- The *cofactor* of $f(x_1,x_2,\ldots,x_i,\ldots,x_n)$ with respect to variable x_i' is $f_{xi'} = f(x_1,x_2,\ldots,x_i=0,\ldots,x_n)$
- Theorem: Shannon's Expansion

$$Let\ f : B^n \rightarrow B.Then\quad f(x_1,x_2,\ldots,x_i,\ldots,x_n) = x_i.f_{x_i} + x_i'.f_{x_i'}$$

$$= (x_i + f_{x_i'}).(x_i' + f_{x_i}) \;\forall \mathrm{i} = 1,2,\ldots,\mathrm{n}$$

- Any function can be expressed as *sum of products* (*product of sums*) of n literals, minterms (maxterms), by recursive expansion.

A sample of Shanon

When we apply Shannon's expansion to a,

$$F(a,b,c,d,e,f) = (a+b+d)(a'+b+c)(b+c+d+e+f)$$

$$F(0,b,c,d,e,f) = (b+d)\cdot 1\cdot(b+c+d+e+f) = (b+d)(1+c+e+f) = b+d$$

$$F(1,b,c,d,e,f) = 1\cdot(b+c)\cdot(b+c+d+e+f) = (b+c)(1+d+e+f) = b+c$$

$$F(a,b,c,d,e,f) = \big(a+F(0,b,c,d,e,f)\big)\cdot\big(a'+F(1,b,c,d,e,f)\big)$$

$$= (a+b+d)(a'+b+c)$$

Definitions

- Another example: f = ab + ac + bc
  - f_a = b + c
  - $f_{a'}$ = bc
  - F = a f_a + a' $f_{a'}$ = a (b + c) + a' (bc)
- A Boolean function can be interpreted as the set of its minterms.
- Operations and relations on Boolean functions can be viewed as operations on their minterm sets
  - Sum of two functions is the Union ($\cup$) of their minterm sets
  - Product of two functions is the Intersection ($\cap$) of their minterm sets
  - Implication between two functions corresponds to containment ($\subseteq$) of their minterm sets
    - $f_1 \rightarrow f_2 \equiv f_1 \subseteq f_2 \equiv f_1' + f_2 = 1$

Boolean Expansion Based on Orthonormal Basis

- Let ϕ_i , i=1,2, …,k be a set of Boolean functions such that $\Sigma_{i=1 \text{ to } k} \ \phi_i = 1$ and $\phi_i \cdot \phi_j = 0$ for $\forall$ i $\neq$ j $\in$ {1,2,…,k}.
- An Orthonormal Expansion of a function f is

  $f = \Sigma_{i=1 \text{ to } k} \ f_{\phi i} \cdot \phi_i$
- $f_{\phi i}$ is called the cofactor of f w.r.t. ϕ_i $\forall$ i.
- Example: f = ab+ac+bc; ϕ_1 = a; ϕ_2 = a';
  - $f_{\phi 1}$ = b+c+bc=b+c
  - $f_{\phi 2}$ = bc
  - f = $\phi_1 \ f_{\phi 1} + \phi_2 \ f_{\phi 2}$ = a (b+c) + a'(bc)=ab+ac+a'bc= ab+ac+bc

Boolean Expansion Based on Orthonormal Basis

- Theorem
  - Let f, g, be two Boolean functions expanded with the same orthonormal basis ϕ_i, i=1,2, …,k
  - Let $f \otimes g = \sum_{i=1}^{k} \Phi_i . (f_{\Phi_i} \otimes g_{\Phi_i})$ on two Boolean functions

- Corollary
  - Let f, g, be two Boolean functions with support variables $\{x_i, i=1,2, \ldots,n\}$.

$$f \otimes g = x_i . (f_{x_i} \otimes g_{x_i}) + x_i' . (f_{x_i'} \otimes g_{x_i'})$$ functions

Boolean Expansion Based on Orthonormal Basis

- Example:
  - Let f = ab + c; g=a'c + b; Compute f $\oplus$g
  - Let ϕ_1=a'b'; ϕ_2=a'b; ϕ_3=ab'; ϕ_4=ab;
  - $f_{\phi 1}$ = c; $f_{\phi 2}$ = c; $f_{\phi 3}$ = c; $f_{\phi 4}$ = 1;
  - $g_{\phi 1}$ = c; $g_{\phi 2}$ = 1; $g_{\phi 3}$ = 0; $g_{\phi 4}$ = 1;
  - f = a'b' (c $\oplus$c) + a'b (c $\oplus$1) + ab' (c $\oplus$0) + ab (1 $\oplus$1)
    = a'bc' + ab'c
  - F= (ab+c) $\oplus$ (a'c+b)= (ab+c)(a+c')b' + (a'+b')c'(a'c+b)
    = (ab+ac)b' + (a'c+a'b)c' = ab'c +a'bc'

Sum of Product (SOP) Simplification Procedure

- 1. Identify all prime implicants covering 1's
  - Example: For a function of 3 variables, group all possible groups of 4, then groups of 2 that are not contained in groups of 4, then minterms that are not contained in a group of 4 or 2.
- 2. Identify all essential prime implicants and select them.
- 3. Check all minterms (1's) covered by essential prime implicants
- 4. Repeat until all minterms (1's) are covered:
  - Select the prime implicant covering the largest uncovered minterms (1's).

Don’t Care Conditions

- In some cases, the function is not specified for certain combinations of input variables as 1 or 0.
- There are two cases in which it occurs:
  - 1. The input combination never occurs.
  - 2. The input combination occurs but we do not care what the outputs are in response to these inputs because the output will not be observed.
- In both cases, the outputs are called as unspecified and the functions having them are called as incompletely specified functions.
- In most applications, we simply do not care what value is assumed by the function for unspecified minterms.

Don’t Care Conditions

- Unspecified minterms of a function are called as don’t care conditions. They provide further simplification of the function, and they are denoted by X’s to distinguish them from 1’s and 0’s.
- In choosing adjacent squares to simplify the function in a map, the don’t care minterms can be assumed either 1 or 0, depending on which combination gives the simplest expression.
- A don’t care minterm need not be chosen at all if it does not contribute to produce a larger implicant.

SOP Simplification Procedure using Don't Cares

- 1. Identify all prime implicants covering 1's & X's
  - Each prime implicant must contain at least a single 1
- 2. Identify all essential prime implicants and select them.
  - An essential prime implicant must be the only implicant covering at least a 1.
- 3. Check all 1's covered by essential prime implicants
- 4. Repeat until all 1's are covered:
  - Select the prime implicant covering the largest uncovered 1's.

Combinational Circuits Design Procedure

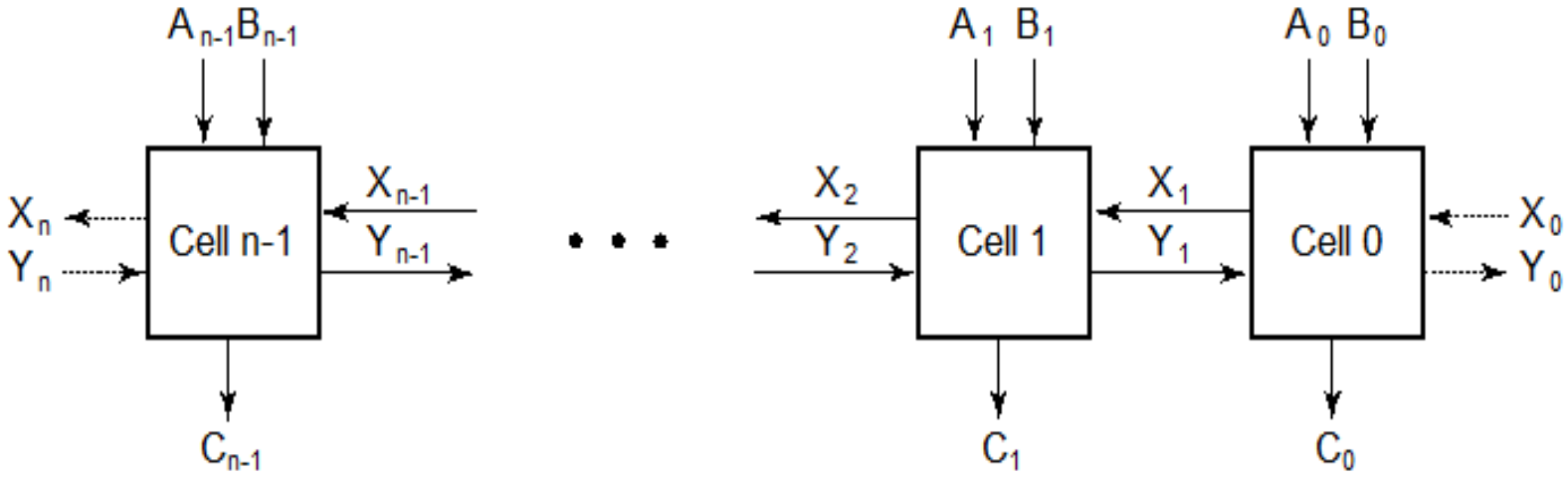
- 1. Specification (Requirement)
  - Write a specification for what the circuit should do e.g. add two 4-bit binary numbers
  - Specify names for the inputs and outputs
- 2. Formulation
  - Convert the Specification into a form that can be Optimized
  - Usually as a truth table or a set of Boolean equations that define the required relationships between the inputs and outputs
- 3. Logic Optimization
  - Apply logic optimization (2-level & multi-level) to minimize the logic circuit
  - Provide a logic diagram or a netlist for the resulting circuit using ANDs, ORs, and inverters

Combinational Circuits Design Procedure

- 4. Technology Mapping and Design Optimization
  - Map the logic diagram or netlist to the implementation technology and gate type selected, e.g. CMOS NANDs
  - Perform design optimizations of gate costs, gate delays, fan-outs, power consumption, etc.
  - Sometimes this stage is merged with stage 3
- 5. Verification
  - Verify that the final design satisfies the original specification- Two methods:
    - Manual: Ensure that the truth table for the final technology-mapped circuit is identical to the truth table derived from specifications
    - By Simulation: Simulate the final technology-mapped circuit on a CAD tool and test it to verify that it gives the desired outputs at the specified inputs and meets delay specs etc.

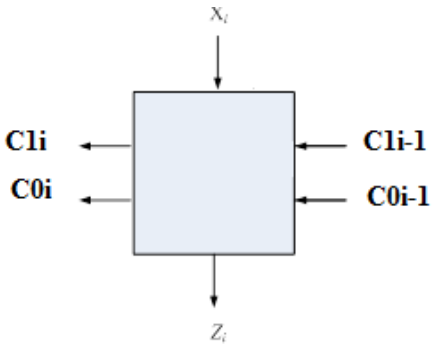
Iterative (Repetitive) Arithmetic Combinational Circuits

- An iterative array can be in a single dimension (1D) or multiple dimensions (spatially)
- Iterative array takes advantage of the regularity to make design feasible
- Block Diagram of a 1D Iterative Array



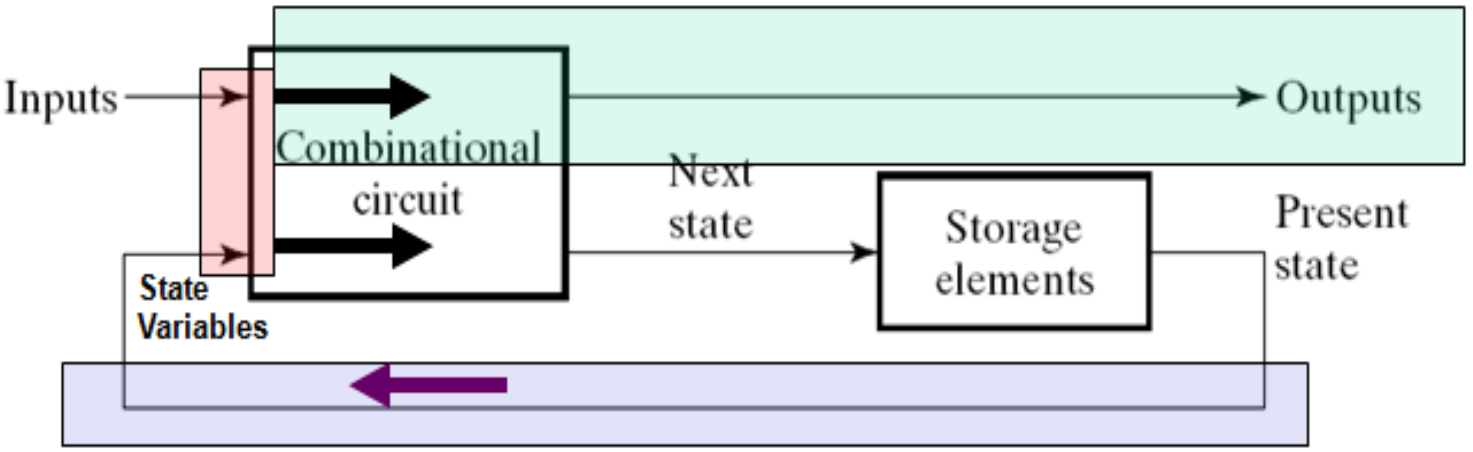
Iterative Design Example

- It is required to design a combinational circuit that computes the equation Y=3*X-1, where X is an n-bit signed 2's complement number
- This circuit can be designed by assuming that we have a borrow feeding first cell or by representing -1 in 2's complement as 11…11 and adding this 1 in each cell.
- We will follow the second approach. We need to represent carry-out values in the range 0 to 3. Thus, we need two signals to represent Carry out values.



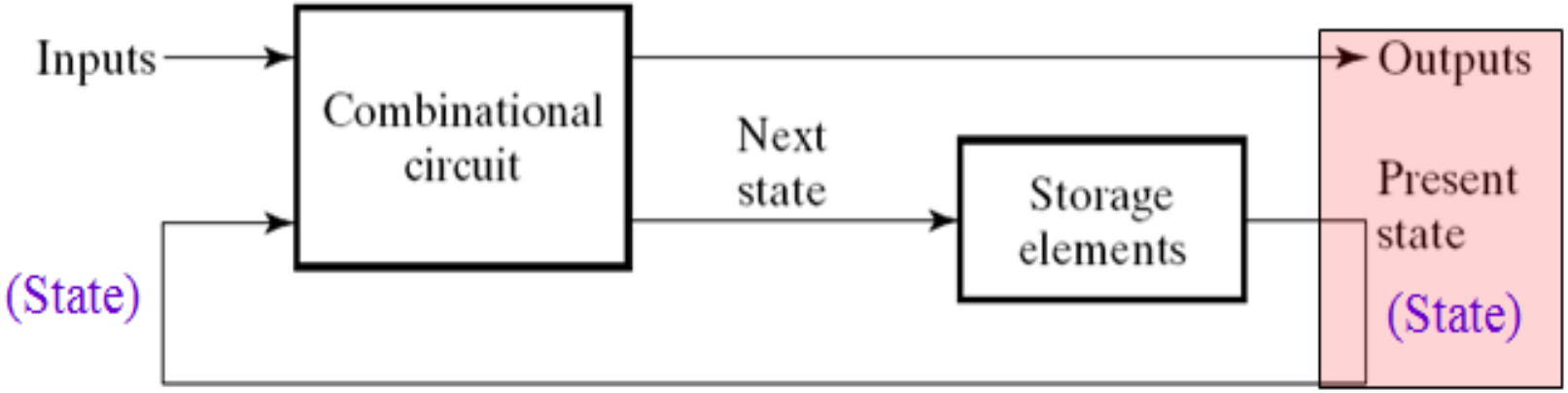
Sequential Circuit Model

- A Sequential circuit consists of:
  - Data Storage elements: (Latches / Flip-Flops)
  - Combinatorial Logic:
    - Implements a multiple-output function
    - Inputs are signals from the outside
    - Outputs are signals to the outside
    - State inputs (Internal): Present State from storage elements
    - State outputs, Next State are inputs to storage elements



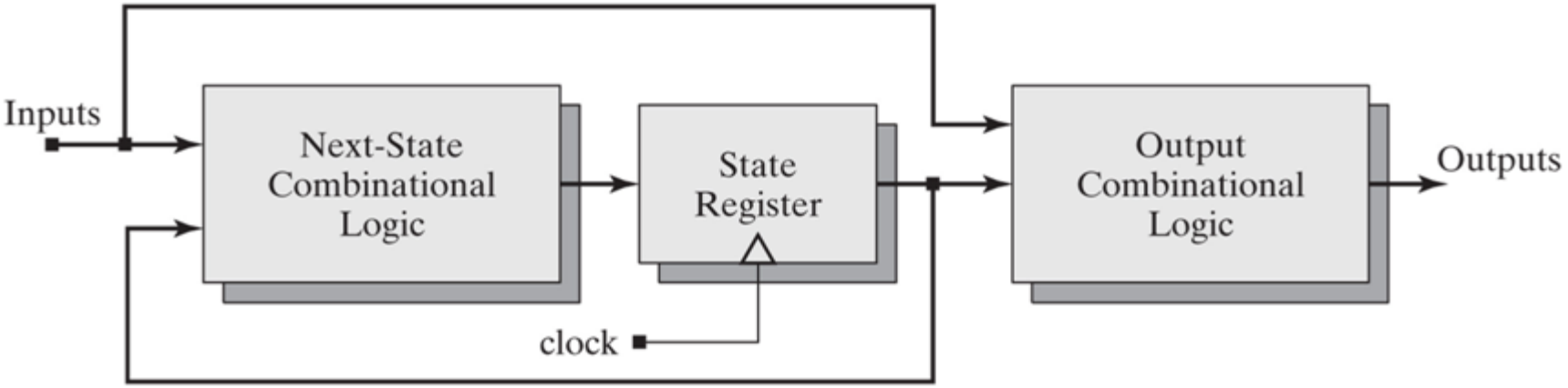
Sequential Circuit Model

- Combinatorial Logic
  - Next state function: Next State = f(Inputs, State)
  - 2 output function types : Mealy & Moore
  - Output function: Mealy Circuits Outputs = g(Inputs, State)
  - Output function: Moore Circuits Outputs = h(State)
- Output function type depends on specification and affects the design significantly

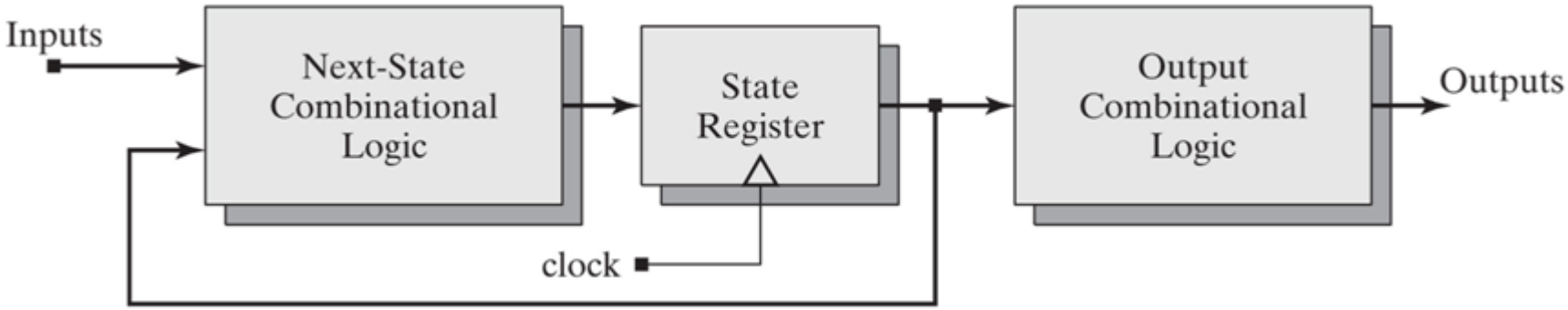


Sequential Circuit Model

Mealy Circuit

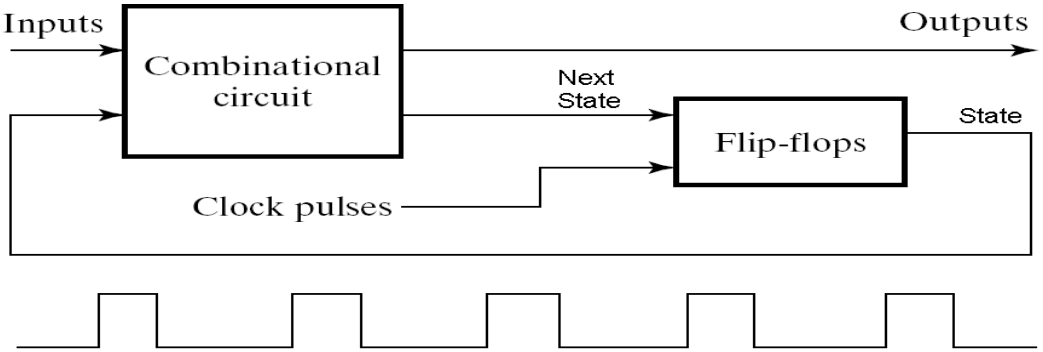


Moore Circuit



Timing of Sequential Circuits
Two Approaches

- Behavior depends on the times at which storage elements 'see' their inputs and change their outputs (next state → present state)
- Asynchronous
  - Behavior defined from knowledge of inputs at any instant of time and the order in continuous time in which inputs change
- Synchronous
  - Behavior defined from knowledge of signals at discrete instances of time
  - Storage elements see their inputs and change state only in relation to a timing signal (clock pulses from a clock)
  - The sy[illegible]lesigns!

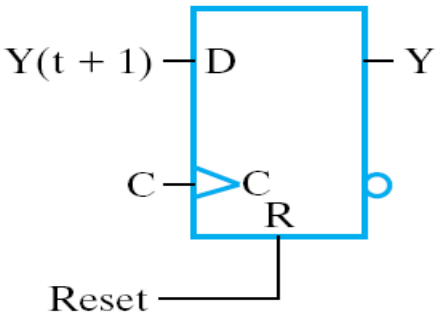


Sequential Circuit Design Procedure

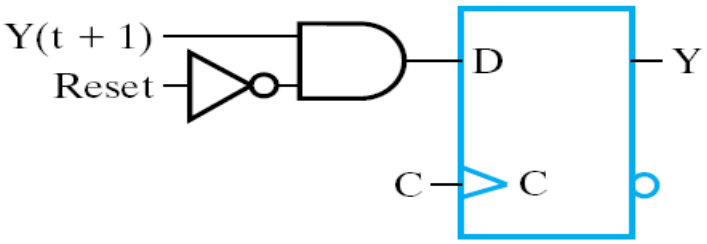
- 1. Specification – e.g. Verbal description
- 2. Formulation – Interpret the specification to obtain a state diagram and a state table
- 3. State Assignment - Assign binary codes to symbolic states
- 4. Flip-Flop Input Equation Determination - Select flip-flop types and derive flip-flop input equations from next state entries in the state table
- 5. Output Equation Determination - Derive output equations from output entries in the state table
- 6. Verification - Verify correctness of final design

State Initialization

- When a sequential circuit is turned on, the state of the flip flops is unknown (Q could be 1 or 0)
- Before meaningful operation, we usually bring the circuit to an initial known state, e.g. by resetting all flip flops to 0's
- This is often done asynchronously through dedicated direct S/R inputs to the FFs
- It can also be done synchronously by going through the clocked FF inputs



(a) Asynchronous Reset



(b) Synchronous Reset

State Minimization

- Aims at reducing the number of machine states
  - reduces the size of transition table.
- State reduction may reduce
  - the number of storage elements.
  - the combinational logic due to reduction in transitions
- Completely specified finite-state machines
  - No don't care conditions.
  - Easy to solve.
- Incompletely specified finite-state machines
  - Unspecified transitions and/or outputs.
  - Intractable problem.

State Minimization for Completely-Specified FSMs

- Equivalent states
  - Given any input sequence the corresponding output sequences match.
- Theorem: Two states are equivalent iff
  - 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.

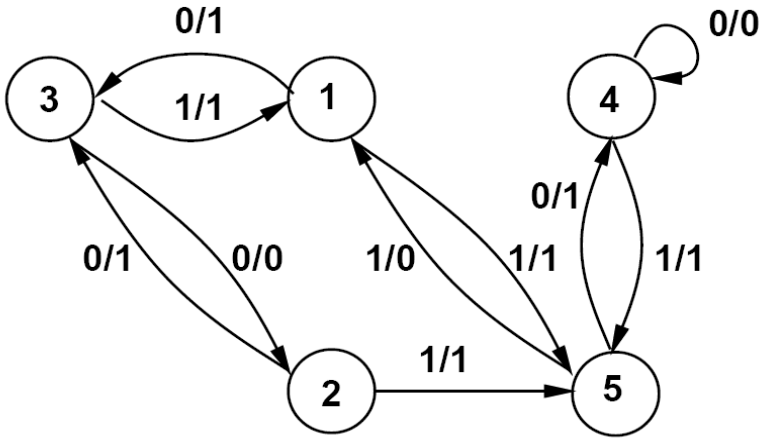
State Minimization Algorithm

- Stepwise partition refinement.
- Initially
  - $\prod_1$ = States belong to the same block when outputs are the same for any input.
- Refine partition blocks: While further splitting is possible
  - $\prod_{k+1}$ = States belong to the same block if they were previously in the same block and their next-states are in the same block of $\prod_k$ for any input.
- At convergence
  - Blocks identify equivalent states.

State Minimization Example

- $\prod_1$ = {(s1, s2), (s3, s4), (s5)}.
- $\prod_2$ = {(s1, s2), (s3), (s4), (s5)}.
- $\prod_2$ = is a partition into equivalence classes
  - States (s1, s2) are equivalent.

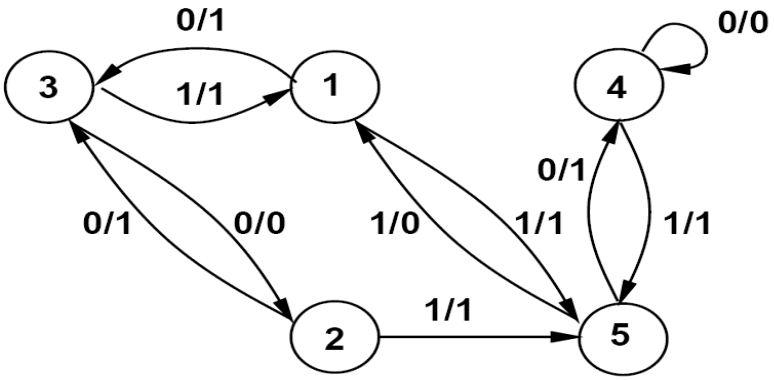
INPUT	STATE	N-STATE	OUTPUT
0	s_1	s_3	1
1	s_1	s_5	1
0	s_2	s_3	1
1	s_2	s_5	1
0	s_3	s_2	0
1	s_3	s_1	1
0	s_4	s_4	0
1	s_4	s_5	1
0	s_5	s_4	1
1	s_5	s_1	0



State Minimization Example

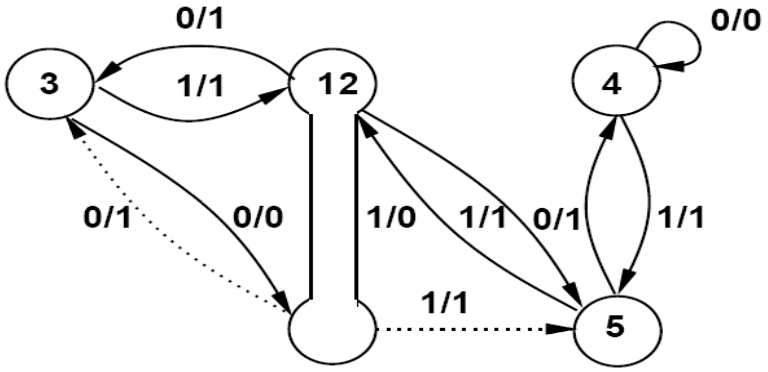
Original FSM

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



Minimal FSM

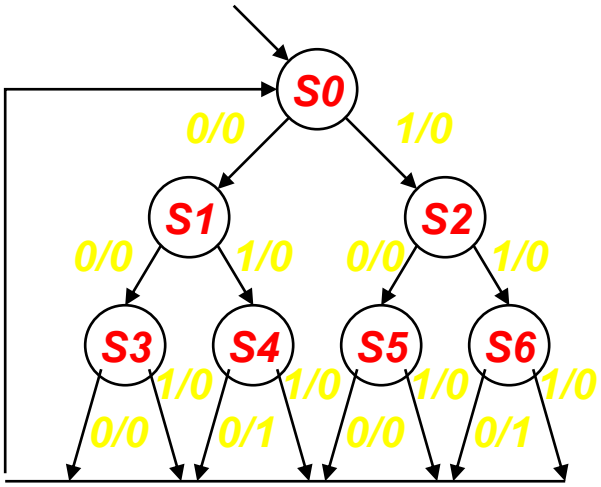
INPUT	STATE	N-STATE	OUTPUT
0	s_{12}	s_3	1
1	s_{12}	s_5	1
0	s_3	s_{12}	0
1	s_3	s_{12}	1
0	s_4	s_4	0
1	s_4	s_5	1
0	s_5	s_4	1
1	s_5	s_{12}	0



Another State Minimization Example

- Sequence Detector for codes of symbols 010 or 110 assuming that each symbol code is 3 bits in length

Input Sequence	Present State	Next State X=0	Next State X=1	Output X=0	Output X=1
Reset	S0	S1	S2	0	0
0	S1	S3	S4	0	0
1	S2	S5	S6	0	0
00	S3	S0	S0	0	0
01	S4	S0	S0	1	0
10	S5	S0	S0	0	0
11	S6	S0	S0	1	0



Another State Minimization Example

Input Sequence	Present State	Next State X=0	Next State X=1	Output X=0	Output X=1
Reset	S0	S1	S2	0	0
0	S1	S3	S4	0	0
1	S2	S5	S6	0	0
00	S3	S0	S0	0	0
01	S4	S0	S0	1	0
10	S5	S0	S0	0	0
11	S6	S0	S0	1	0

(S0 S1 S2 S3 S4 S5 S6)

(S0 S1 S2 S3 S5) (S4 S6)

(S0 S3 S5) (S1 S2) (S4 S6)

(S0) (S3 S5) (S1 S2) (S4 S6)

S1 is equivalent to S2

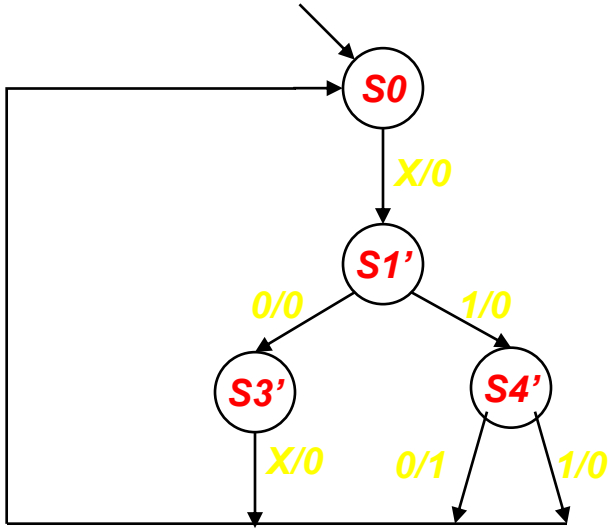
S3 is equivalent to S5

S4 is equivalent to S6

Another State Minimization Example

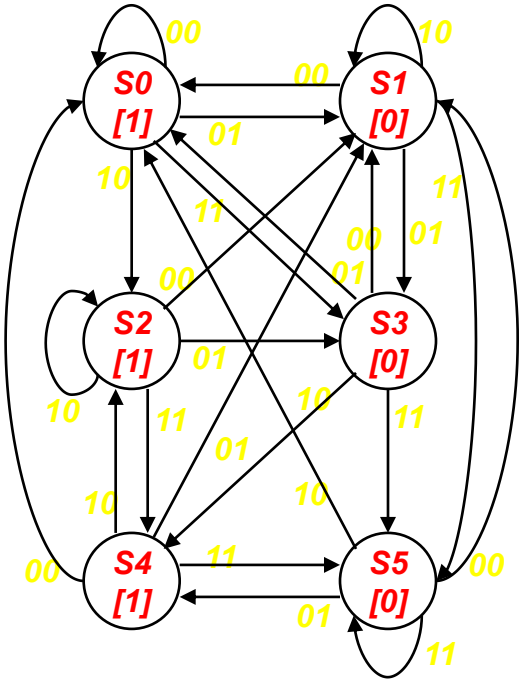
- State minimized sequence detector for 010 or 110

Input Sequence	Present State	Next State X=0	Next State X=1	Output X=0	Output X=1
Reset	S0	S1'	S1'	0	0
0 + 1	S1'	S3'	S4'	0	0
X0	S3'	S0	S0	0	0
X1	S4'	S0	S0	1	0



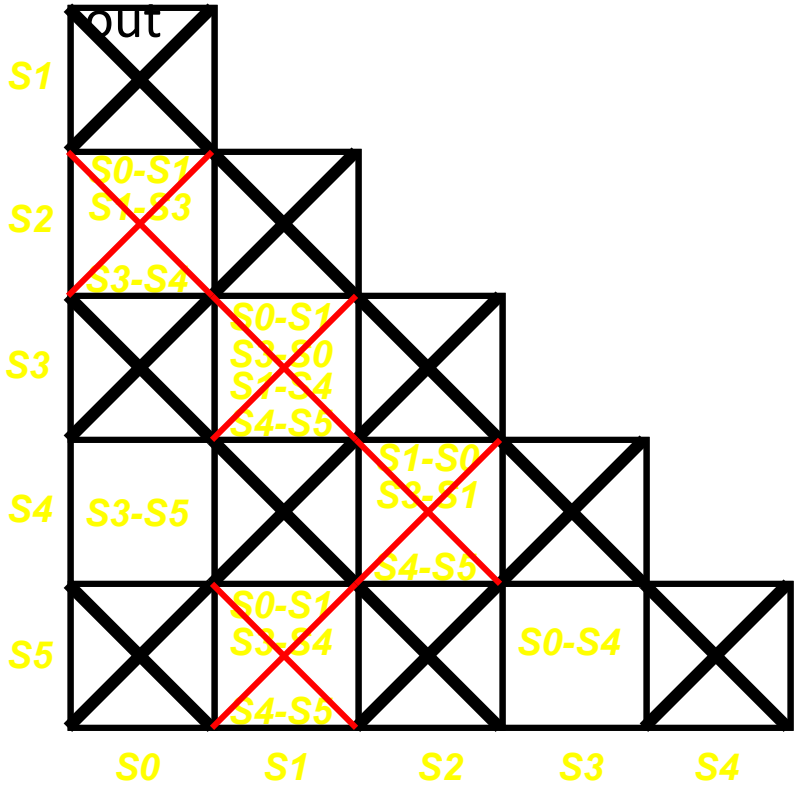
Multiple Input Example

present state	next state 00	01	10	11	output
S0	S0	S1	S2	S3	1
S1	S0	S3	S1	S4	0
S2	S1	S3	S2	S4	1
S3	S1	S0	S4	S5	0
S4	S0	S1	S2	S5	1
S5	S1	S4	S0	S5	0



Implication Chart Method

- Cross out incompatible states based on outputs
- Then cross out more cells if indexed chart entries are already crossed out



present state	next state 00	01	10	11	output
S0	S0	S1	S2	S3	1
S1	S0	S3	S1	S4	0
S2	S1	S3	S2	S4	1
S3	S1	S0	S4	S5	0
S4	S0	S1	S2	S5	1
S5	S1	S4	S0	S5	0

present state	next state 00	01	10	11	output
S0'	S0'	S1	S2	S3'	1
S1	S0'	S3'	S1	S0'	0
S2	S1	S3'	S2	S0'	1
S3'	S1	S0'	S0'	S3'	0

minimized state table (S0==S4) (S3==S5)

State Minimization Computational Complexity

- Polynomially-bound algorithm.
- There can be at most |S| partition refinements.
- Each refinement requires considering each state
  - Complexity $O(|S|^2)$.
- Actual time may depend upon
  - Data-structures.
  - Implementation details.

State Encoding

- Determine a binary encoding of the states ($|S|=n_s$) that optimize machine implementation
  - Area
  - Cycle-time
  - Power dissipation
  - Testability
- Assume D-type registers.
- Circuit complexity is related to
  - Number of storage bits n_b used for state representation
  - Size of combinational component
- There are $2^{n_b}!/(2^{n_b}-n_s)!$ ssible encodings
- Implementation Modeling
  - Two-level circuits.
  - Multiple-level circuits.

State Encoding Example

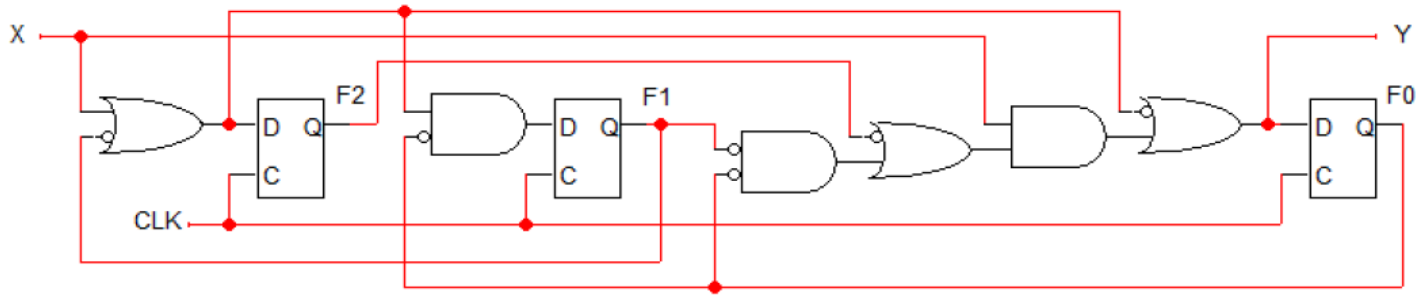
Table 1. An example of an FSM.

Present State	Next State		Output	
	X=0	X=1	X=0	X=1
S0	S2	S2	0	0
S1	S2	S0	0	1
S2	S4	S3	0	1
S3	S1	S2	1	0
S4	S1	S4	1	0

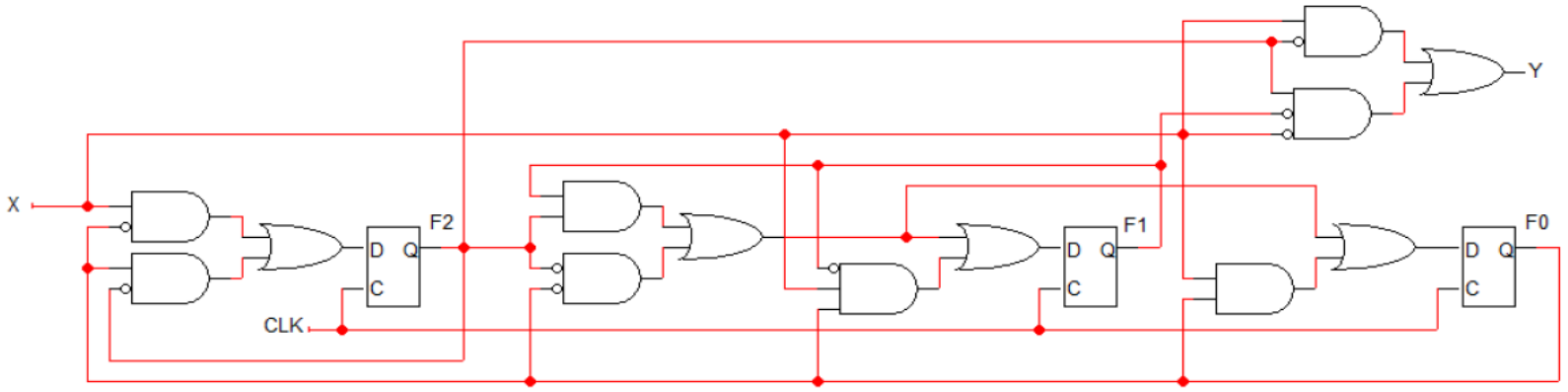
Table 2. State assignments with resulting area cost.

State	Ass. 1	Ass. 2
S0	101	111
S1	001	000
S2	100	011
S3	111	101
S4	110	100
Area (No. of Literals)	10	20

State Encoding Example



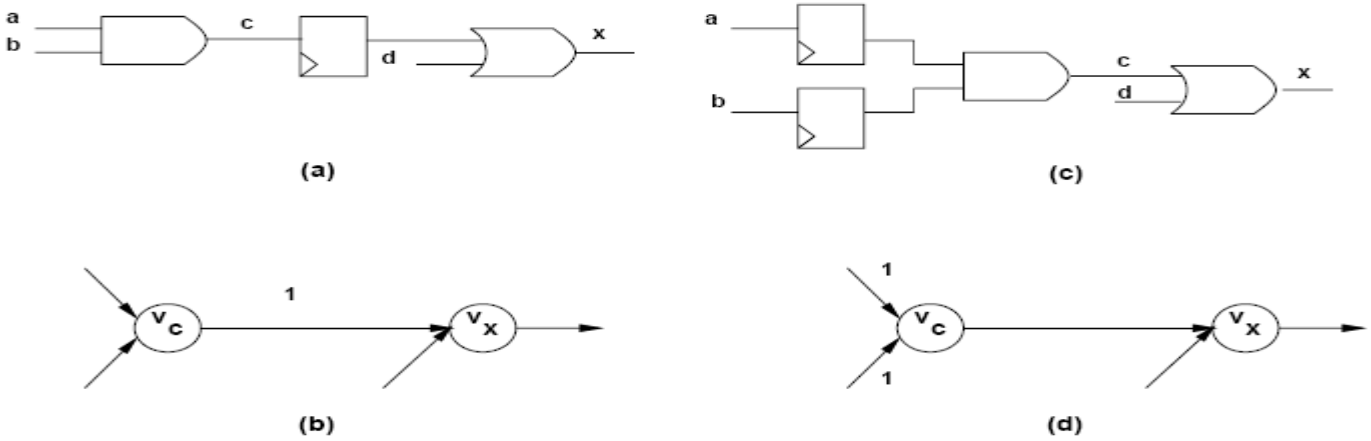
(a) *Circuit resulting from "Ass. 1".*



(b) *Circuit resulting from "Ass. 2".*

Retiming

- Minimize cycle-time or area by changing register positions.
- Do not modify combinational logic.
- Preserve network structure
  - Modify weights.
  - Do not modify graph structure.



Retiming

- Global optimization technique [Leiserson].
- Changes register positions
  - affects area
    - changes register count.
  - affects cycle-time
    - changes path delays between register pairs.
- Solvable in polynomial time.
- Assumptions
  - Vertex delay is constant: No fanout delay dependency.
  - Graph topology is invariant: No logic transformations.
  - Synchronous implementation
    - Cycles have positive weights.
    - Edges have non-negative weights.

Retiming

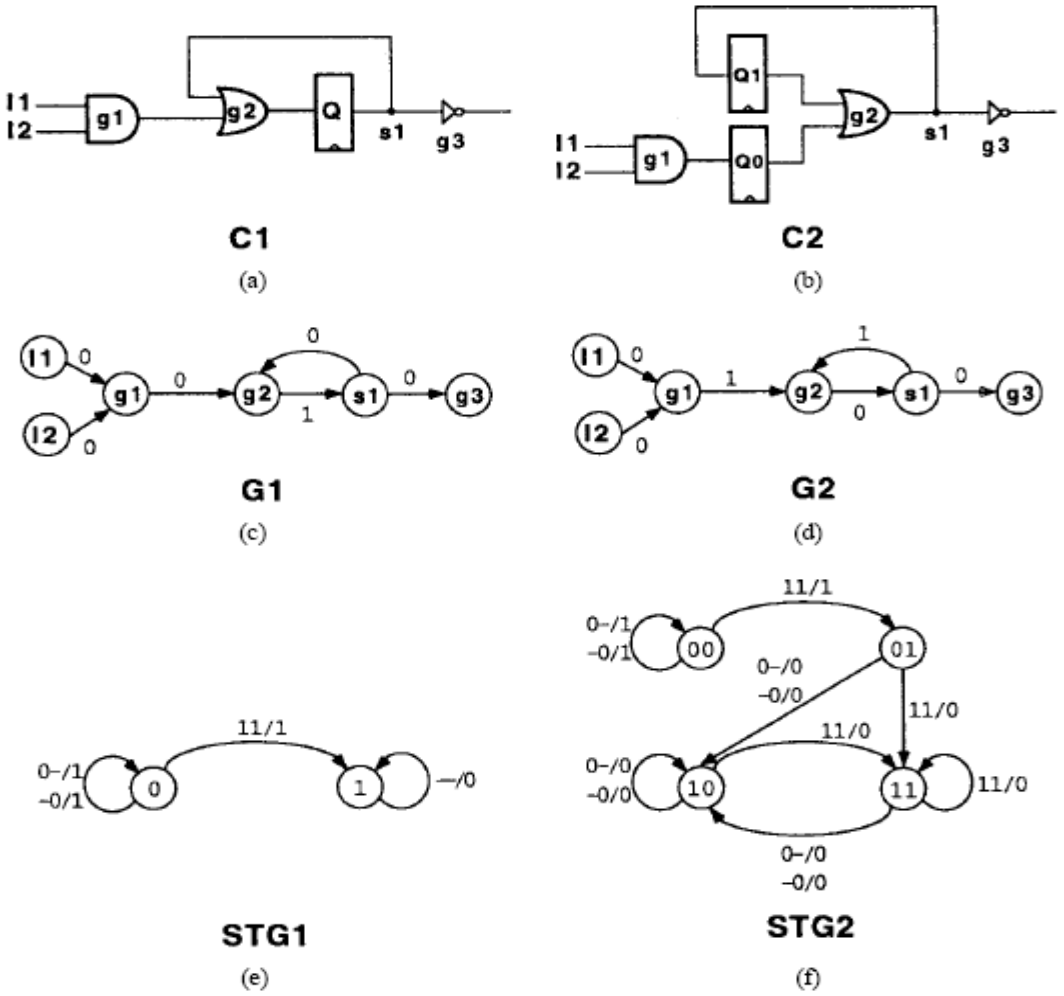


Fig. 2. An example of a backward retiming move across a single-output combinational gate.

Retiming

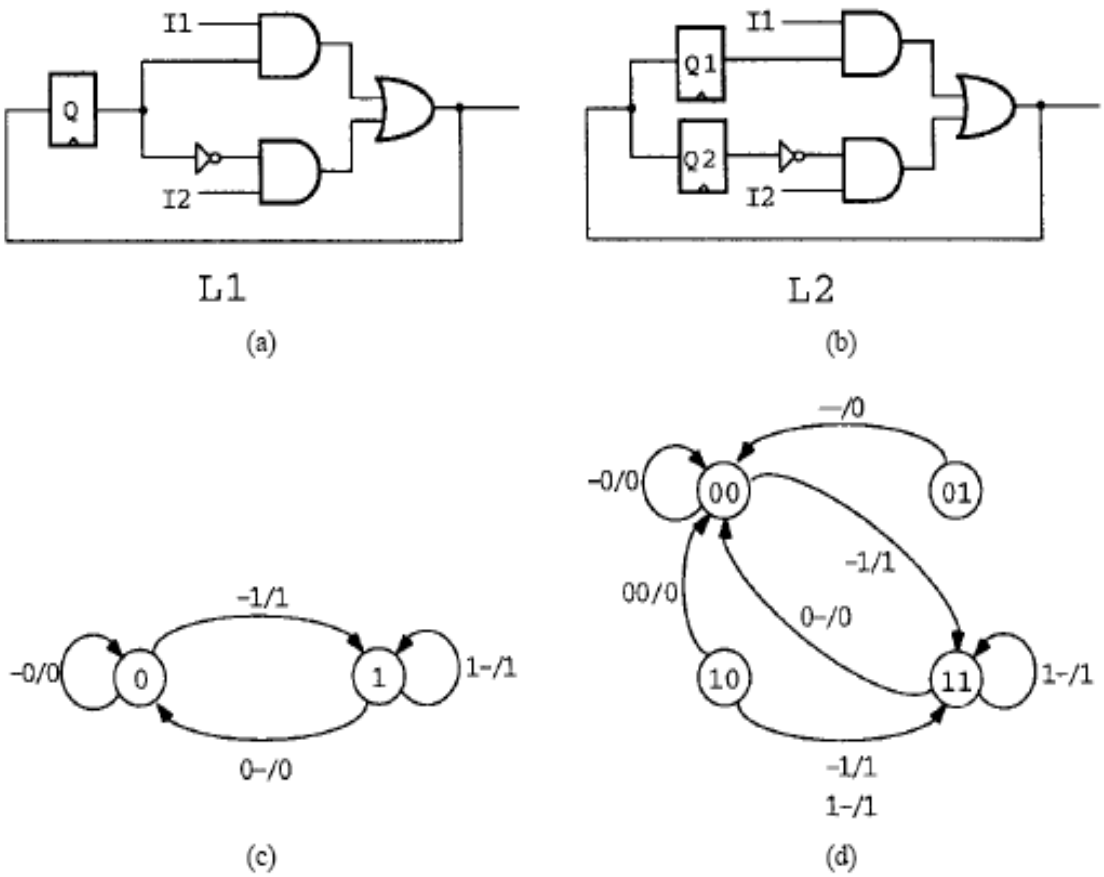
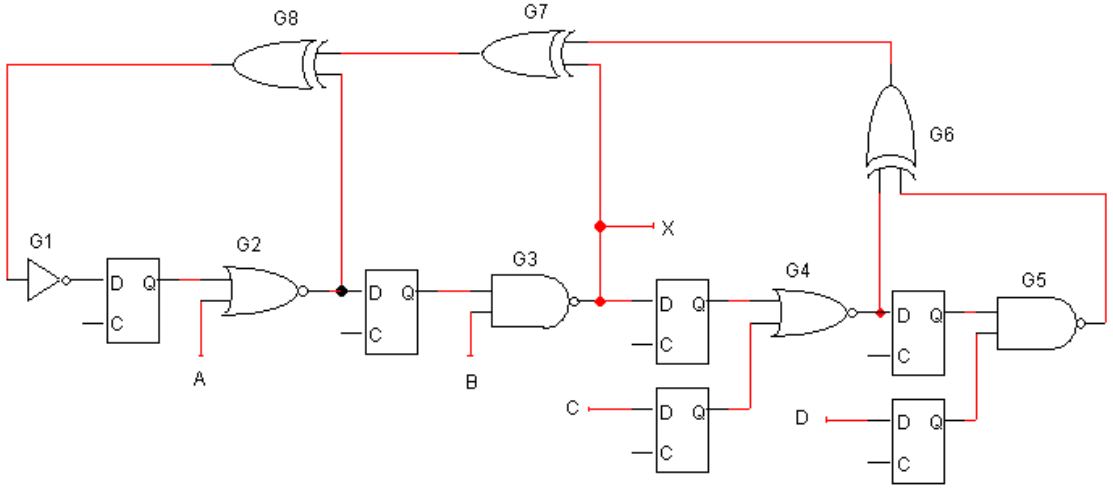


Fig. 3. An example of a forward retiming move across a fanout stem.

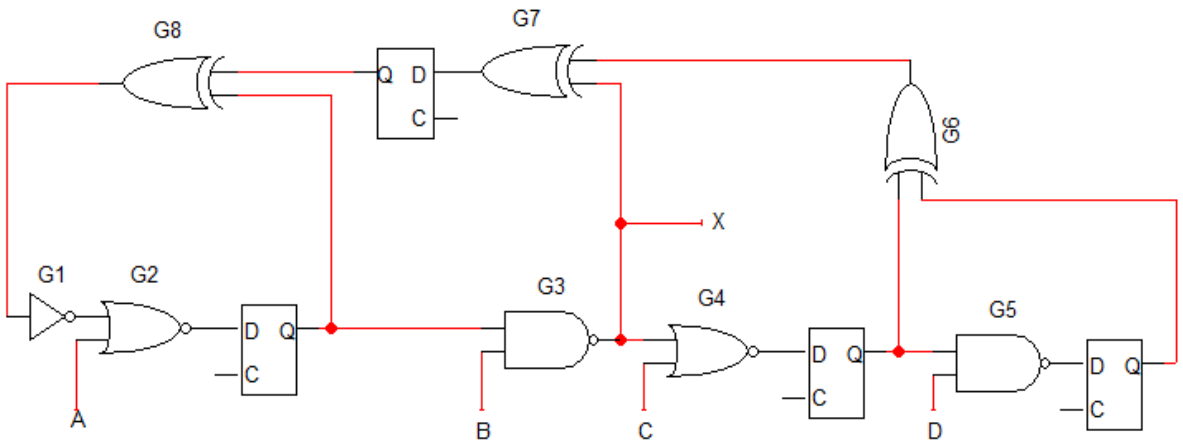
Retiming Example

- Consider the sequential circuit given below having 4 inputs {A, B, C, D} and one output {X}.
- Assume that the delay of an inverter is 1 unit delay, the delay of a 2-input NAND gate is 2 unit delays, the delay of a 2-input NOR gate is 2 unit delays and the delay of a 2-input XOR gate is 3 unit delays.
- Using only the Retiming transformation, minimize the critical path of this circuit with the r

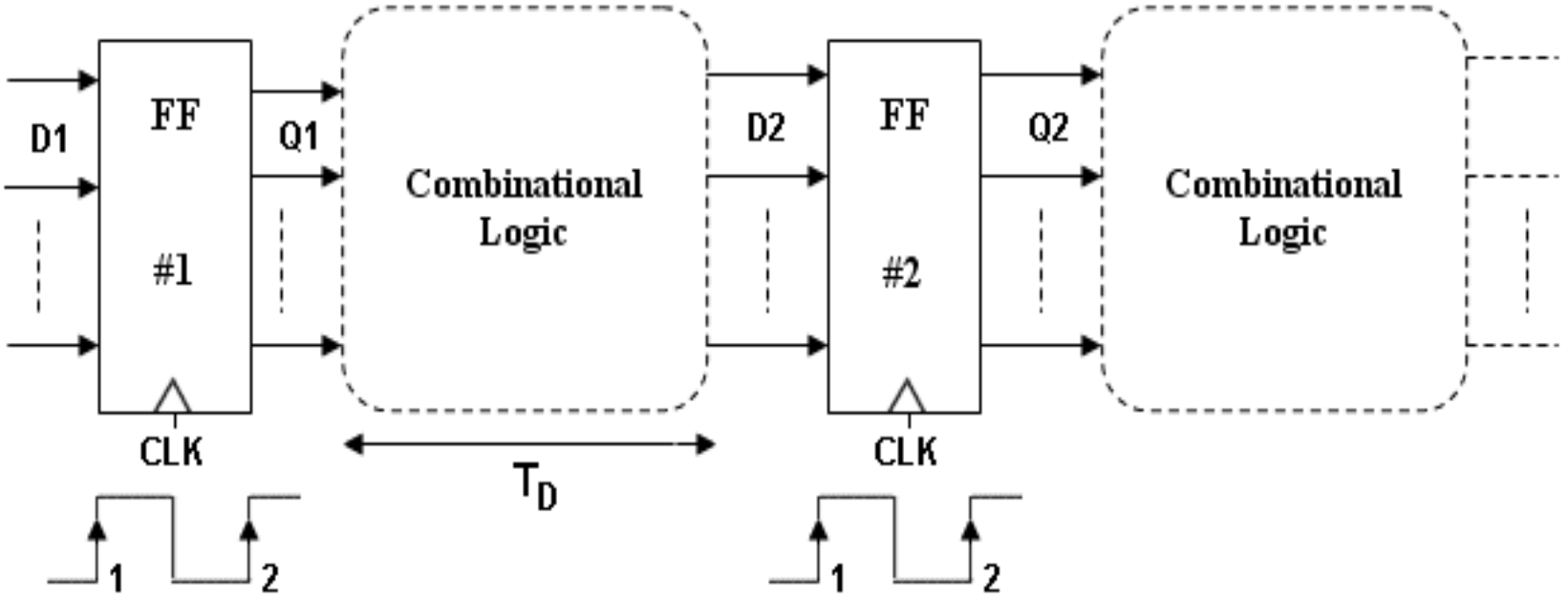


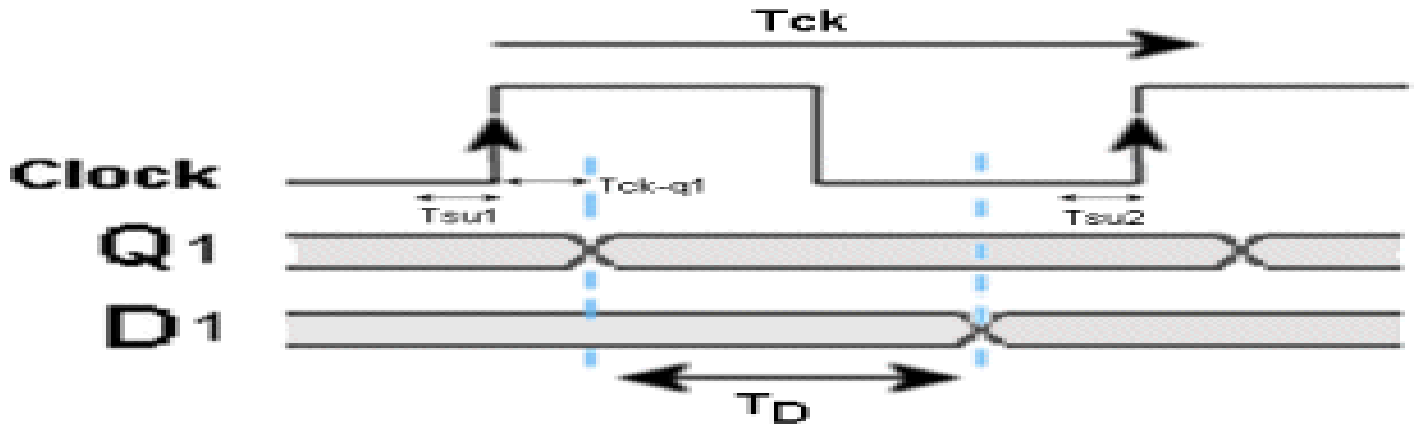
Retiming Example

- The maximum propagation delay is 12 and there are two critical paths as follows:
  - {G5, G6, G7, G8, G1}, {G4, G6, G7, G8, G1},
- We can apply the following retiming transformations to reduce the critical path:
  - Retime G5 by -1 (forward retiming); Retime G4 by -1 (forward retiming); Retime G1 by +1 (backward retiming); Retime G8 by +1 (backward retiming); Retime the stem on fanout of G2 by +1 (backward retiming)
- The resulting retiming circuit is as follows which has a maximum propagation delay of 6 with only 4 FFs.



Sequential Circuit Timing





Timing Constraints

- T_D = worst case delay through combinational logic
- T_{SU} = FF set up time – Minimum time before the clock edge where the input data must be ready and stable
- $T_{clk \rightarrow Q}$ = Clock to Q delay – Time between clock edge and data appearing at the output of the FF
- T_{Hold} = FF hold time – Minimum time after the clock edge where data has to remain stable (held stable)
- Based on the FF & combinational logic timing parameters, the following timing constraints are obtained for correct operation of the circuit: $T_{clk} \geq T_{clk \rightarrow q1} + T_D + T_{su}$

Timing Constraints

- The previous equation assumes that the clock arrives at all FFs, at exactly the same time!
- Clock Skew (T_{skew}) is the delay between clocks at different chip locations.
- To take Clock Skew into account: $T_{clk} \geq T_{clk \rightarrow q1} + T_D + T_{su} + T_{skew}$
- Clock Signals will have random variations in their Periods and Frequencies, called Jitter.
- The latest arrival time minus the earliest arrival time during an observed period of time is called the "peak to peak jitter amplitude".
- We have to take the Peak to Peak Jitter ($T_{P\text{-}P\ Jitter}$) into account

$T_{clk} \geq T_{clk \rightarrow q1} + T_D + T_{su} + T_{skew} + T_{P\text{-}P\ Jitter}$

Timing Constraints

- Another Timing Constraint arises in situations where T_D is "Zero" or very small when the output of a FF is fed directly to the input of another (e.g. in Shift Registers).
- In such situation, we need to make sure that the data does not pass through two FFs (during the transparency window of the FF where both master and slave are enabled).
- Hence to avoid Hold Time violation:

  $T_{skew} + T_{P\text{-}P\ Jitter} + T_{hold2} \leq T_{ck \rightarrow q1} + T_D$

  where T_{hold2} is the hold time of the 2nd FF

Metastability

- Whenever there are setup and hold time violations in any flip-flop, it enters a state where its output is unpredictable: this state is known as metastable state (quasi stable state)
- At the end of metastable state, the flip-flop settles down to either '1' or '0'. This whole process is known as metastability.
- When a flip-flop is in metastable state, its output oscillates between '0' and '1‘. How long it takes to settle down, depends on the technology of the flip-flop.
- Metastability occurs when the input signal is an asynchronous signal.

Metastability

- The most common way to tolerate metastability is to add one or more successive synchronizing flip-flops to the synchronizer.
- This approach allows for an entire clock period (except for the setup time of the second flip-flop) for metastable events in the first synchronizing flip-flop to resolve themselves.
- This does, however, increase the latency in the synchronous logic's observation of input changes.