Chapter #6: Sequential Logic Design

Contemporary Logic Design

Randy H. Katz
University of California, Berkeley

June 1993

Chapter Overview

" ***Sequential Networks***
- Simple Circuits with Feedback
- R-S Latch
- J-K Flipflop
- Edge-Triggered Flipflops

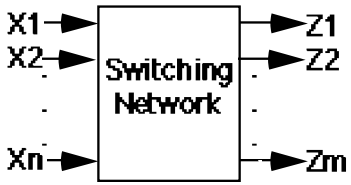
" ***Timing Methodologies***
- Cascading Flipflops for Proper Operation
- Narrow Width Clocking vs. Multiphase Clocking
- Clock Skew

" ***Realizing Circuits with Flipflops***
- Choosing a FF Type
- Characteristic Equations
- Conversion Among Types

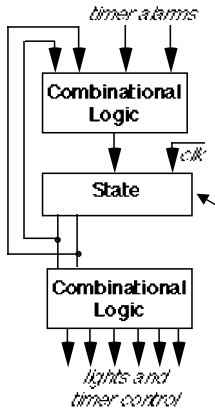
" ***Metastability and Asynchronous Inputs***

" ***Self-Timed Circuits***

Sequential Switching Networks



Circuits with Feedback:
Some outputs are also inputs



Traffic Light Controller is a complex sequential logic network

Sequential logic forms basis for building "memory" into circuits

These memory elements are primitive sequential circuits

Sequential Switching Networks

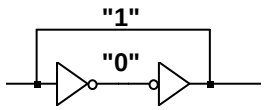
Simple Circuits with Feedback

Primitive memory elements created from cascaded gates

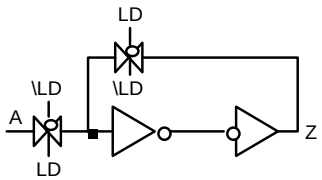
Simplest gate component: inverter

Basis for commercial static RAM designs

Cross-coupled NOR gates and NAND gates also possible



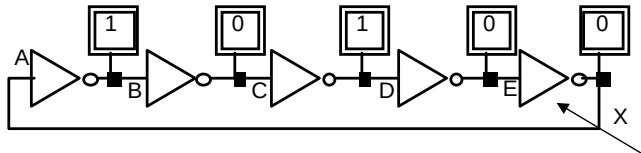
Cascaded Inverters: Static Memory Cell



Selectively break feedback path to load new value into cell

Sequential Switching Networks

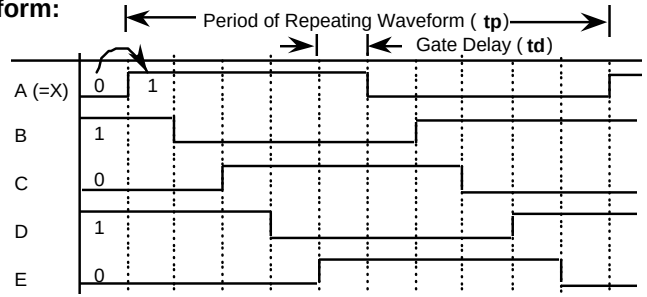
Inverter Chains



Odd # of stages leads to ring oscillator
Snapshot taken just before last inverter changes

Output high propagating thru this stage

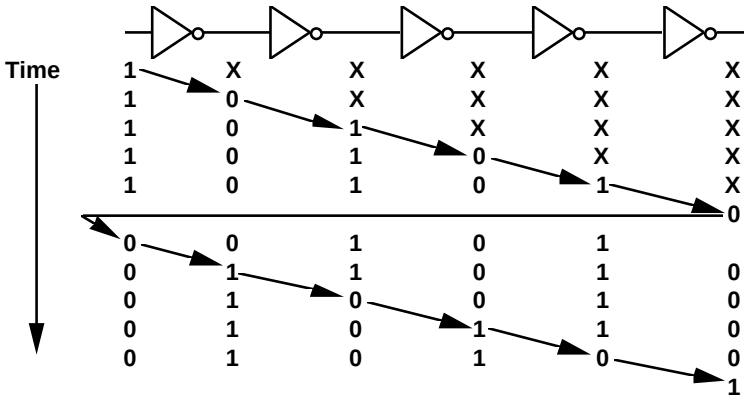
Timing Waveform:



tp = n * td
n = # inverters

Sequential Switching Networks

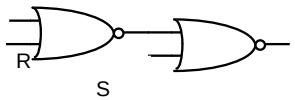
Inverter Chains

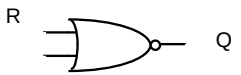


Propagation of Signals through the Inverter Chain

Sequential Switching Networks

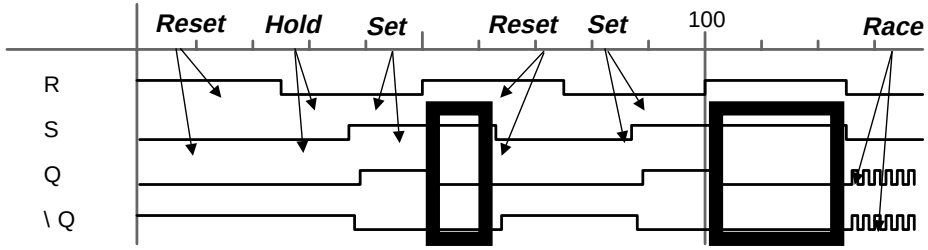
Cross-Coupled NOR Gates





Just like cascaded inverters, with capability to force output to 0 (reset) or 1 (set)

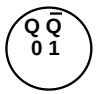
Timing Waveform



Sequential Switching Networks

State Behavior of R-S Latch

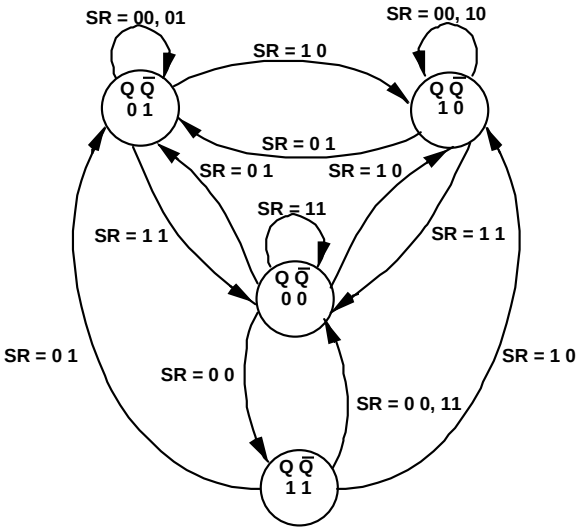
S	R	Q
0	0	hold
0	1	0
1	0	1
1	1	unstable



Truth Table Summary of R-S Latch Behavior

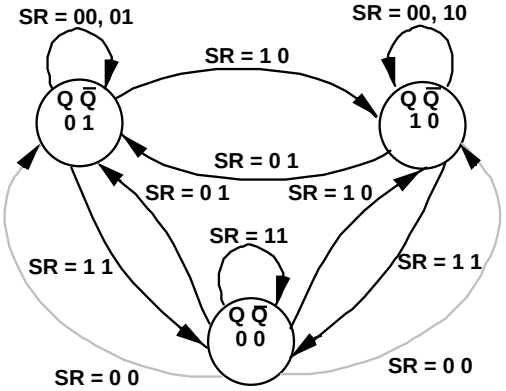
Sequential Logic Networks

Theoretical R-S Latch State Diagram



Sequential Logic Networks

Observed R-S Latch Behavior



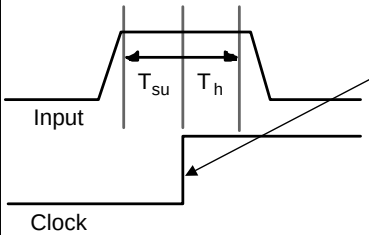
Very difficult to observe R-S Latch in the 1-1 state

Ambiguously returns to state 0-1 or 1-0

A so-called "race condition"

Sequential Switching Networks

Definition of Terms



Clock:
Periodic Event, causes state of memory element to change

rising *edge*, falling *edge*, high *level*, low *level*

Setup Time (Tsu)
Minimum time before the clocking event by which the input must be stable

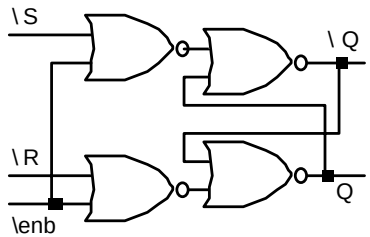
Hold Time (Th)
Minimum time after the clocking event during which the input must remain stable

There is a timing "window" around the clocking event during which the input must remain stable and unchanged in order to be recognized

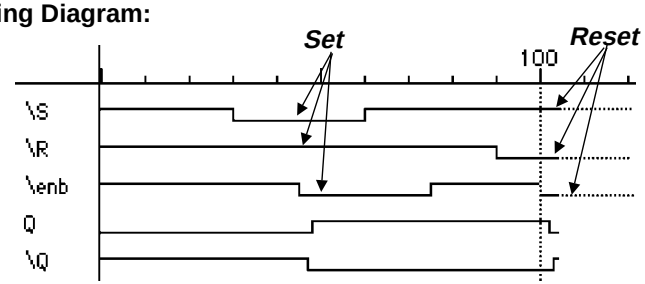
Sequential Switching Networks

Level-Sensitive Latch **aka Gated R-S Latch**

Schematic:



Timing Diagram:



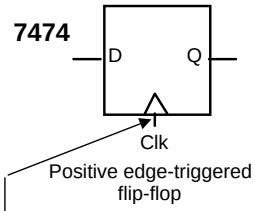
Sequential Switching Networks

Latches vs. Flipflops

Input/Output Behavior of Latches and Flipflops

Type	When Inputs are Sampled	When Outputs are Valid
unclocked latch	always	propagation delay from input change
level sensitive latch	clock high (Tsu, Th around falling clock edge)	propagation delay from input change
positive edge flipflop	clock lo-to-hi transition (Tsu, Th around rising clock edge)	propagation delay from rising edge of clock
negative edge flipflop	clock hi-to-lo transition (Tsu, Th around falling clock edge)	propagation delay from falling edge of clock
master/slave flipflop	clock hi-to-lo transition (Tsu, Th around falling clock edge)	propagation delay from falling edge of clock

Sequential Switching Networks

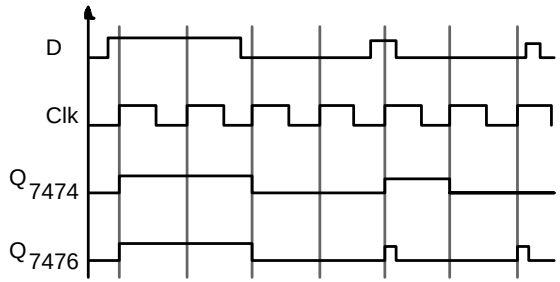


Bubble here for negative edge triggered device

Edge triggered **device sample inputs on the event edge**

Transparent latches **sample inputs as long as the clock is asserted**

Timing Diagram:



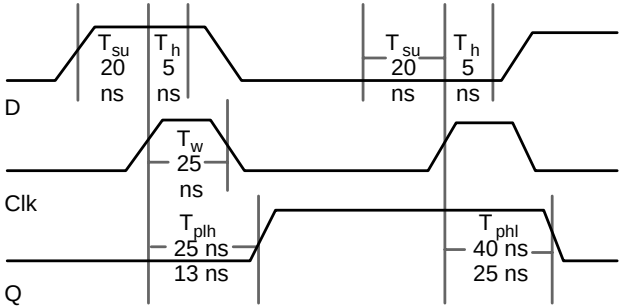
Behavior the same unless input changes while the clock is high

Sequential Switching Elements

Typical Timing Specifications: Flipflops vs. Latches

74LS74 Positive Edge Triggered D Flipflop

" Setup time
" Hold time
" Minimum clock width
" Propagation delays (low to high, high to low, max and typical)



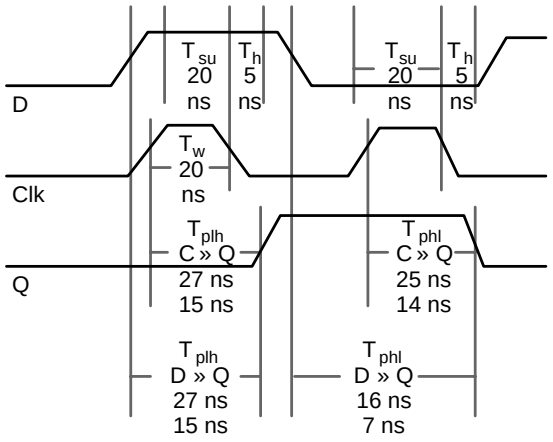
All measurements are made from the clocking event that is, the *rising edge* of the clock

Sequential Switching Networks

Typical Timing Specifications: Flipflops vs. Latches

74LS76 Transparent Latch

" Setup time
" Hold time
" Minimum Clock Width
" Propagation Delays:
high to low, low to high, maximum, typical
data to output
clock to output



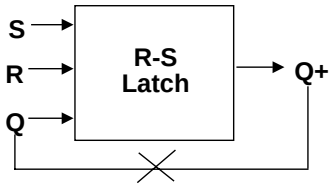
Measurements from falling clock edge or rising or falling data edge

Sequential Switching Elements

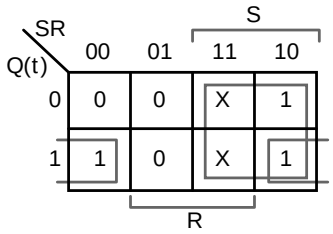
R-S Latch Revisited

Truth Table:
Next State = F(S, R, Current State)

S(t)	R(t)	Q(t)	Q(t+Δ)	
0	0	0	0	
0	0	1	1	HOLD
0	1	0	0	
0	1	1	0	RESET
1	0	0	1	
1	0	1	1	SET
1	1	0	X	
1	1	1	X	Not Allowed



Derived K-Map:



Characteristic Equation:

$$Q+ = S + \bar{R}\, Q_t$$

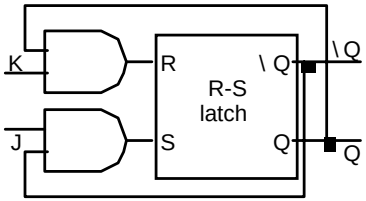
Sequential Switching Networks

J-K Flipflop

How to eliminate the forbidden state?

Idea: use output feedback to guarantee that R and S are never both one

J, K both one yields toggle



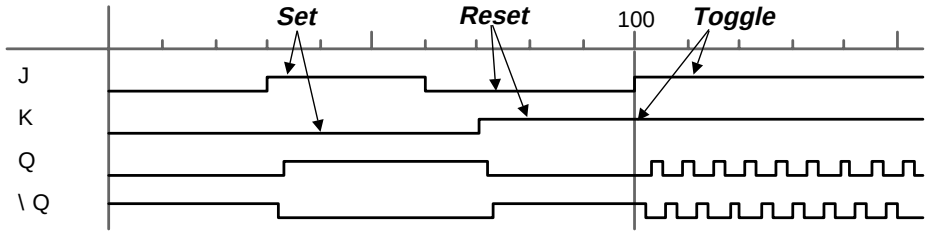
J(t)	K(t)	Q(t)	Q(t+Δ)	
0	0	0	0	
0	0	1	1	HOLD
0	1	0	0	
0	1	1	0	RESET
1	0	0	1	
1	0	1	1	SET
1	1	0	1	
1	1	1	0	TOGGLE

Characteristic Equation:

$$Q+ = Q\,\bar{K} + \bar{Q}\,J$$

Sequential Switching Networks

J-K Latch: Race Condition



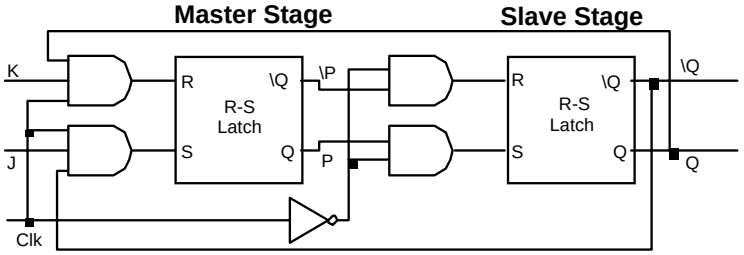
Race Condition

Toggle Correctness: Single State change per clocking event

Solution: Master/Slave Flipflop

Sequential Switching Network

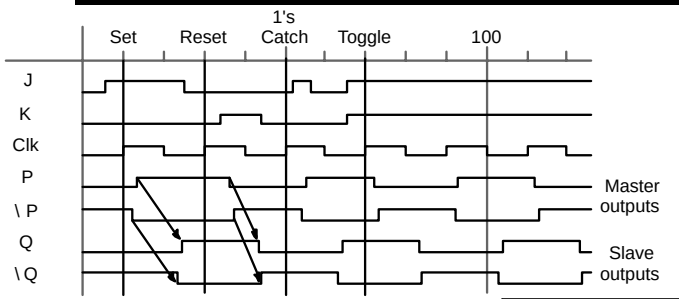
Master/Slave J-K Flipflop



Sample inputs while clock high

Sample inputs while clock low

Uses time to break feedback path from outputs to inputs!



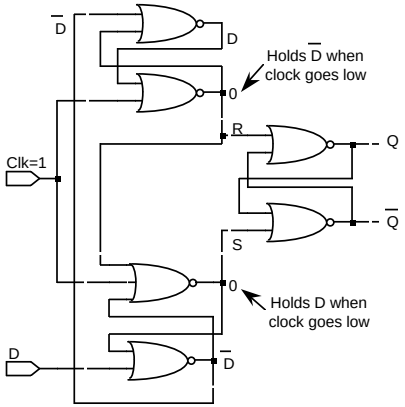
Correct Toggle Operation

Sequential Switching Networks

Edge-Triggered Flipflops

1's Catching: **a 0-1-0 glitch on the J or K inputs leads to a state change! forces designer to use hazard-free logic**

Solution: **edge-triggered logic**



Negative edge-triggered FF when clock is high

Negative Edge-Triggered D flipflop

4-5 gate delays

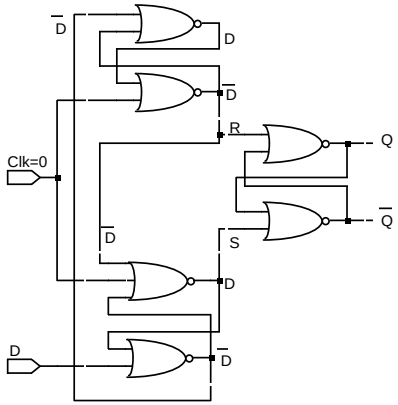
setup, hold times necessary to successfully latch the input

Characteristic Equation:
Q+ = D

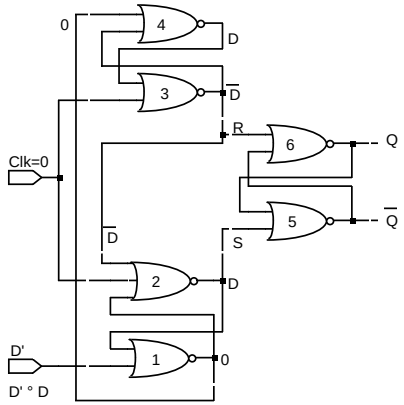
Sequential Switching Network

Edge-triggered Flipflops

Step-by-step analysis



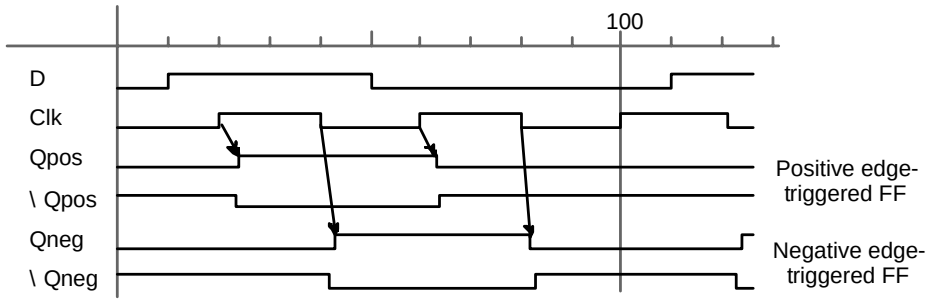
Negative edge-triggered FF when clock goes high-to-low data is latched



Negative edge-triggered FF when clock is low data is held

Sequential Switching Networks

Positive vs. Negative Edge Triggered Devices



Positive Edge Triggered

Inputs sampled on rising edge
Outputs change after rising edge

Negative Edge Triggered

Inputs sampled on falling edge
Outputs change after falling edge

Toggle Flipflop

Formed from J-K with both inputs wired together

Timing Methodology

Overview

" Set of rules for interconnecting components and clocks

" When followed, guarantee proper operation of system

" Approach depends on building blocks used for memory elements

For systems with latches:

Narrow Width Clocking

Multiphase Clocking (e.g., Two Phase Non-Overlapping)

For systems with edge-triggered flipflops:

Single Phase Clocking

" Correct Timing:

(1) correct inputs, with respect to time, are provided to the FFs

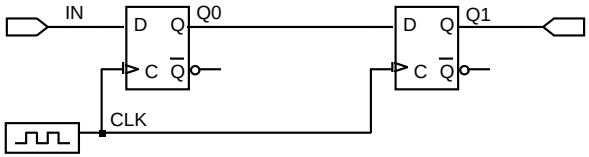
(2) no FF changes more than once per clocking event

Timing Methodologies

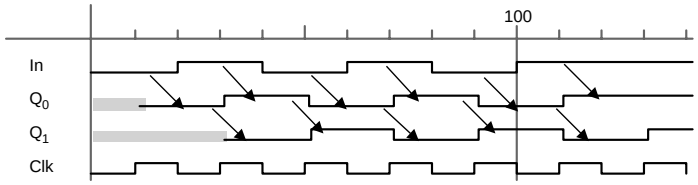
Cascaded Flipflops and Setup/Hold/Propagation Delays

Shift Register
S,R are preset, preclear

New value to first stage while second stage obtains current value of first stage



Correct Operation, assuming positive edge triggered FF



Timing Methodologies

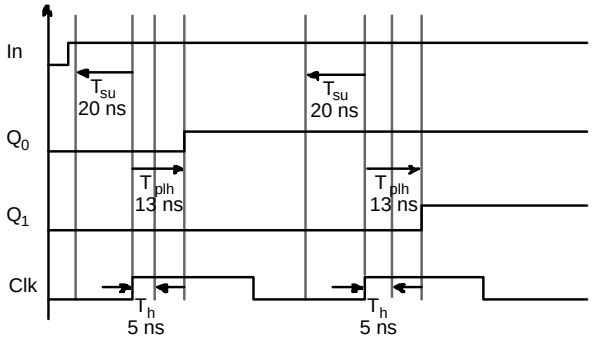
Cascaded Flipflops and Setup/Hold/Propagation Delays

Why this works:

" Propagation delays far exceed hold times; Clock width constraint exceeds setup time

" This guarantees following stage will latch current value before it is replaced by new value

" Assumes infinitely fast distribution of the clock



Timing constraints guarantee proper operation of cascaded components

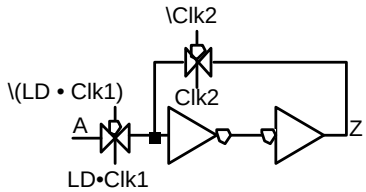
Timing Methodologies

Narrow Width Clocking versus Multiphase Clocking

Level Sensitive Latches vs. Edge Triggered Flipflops

" Latches use fewer gates to implement a memory function

" Less complex clocking with edge triggered devices



CMOS Dynamic Storage Element

Feedback path broken by two phases of the clock (just like master/slave idea!)

8 transistors to implement memory function

but requires two clock signals constrained to be non-overlapping

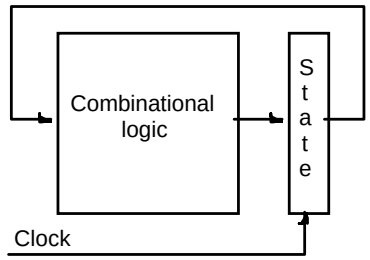
Edge-triggered D-FF: 6 gates (5 x 2-input, 1 x 3-input) = 26 transistors!

Timing Methodologies

Narrow Width Clocking for Systems with Latches for State

Generic Block Diagram for Clocked Sequential System

state implemented by latches or edge-triggered FFs



Two-sided Constraints:
must be careful of very fast signals as well as very slow signals!

Clock Width < fastest propagation through comb. logic plus latch prop delay

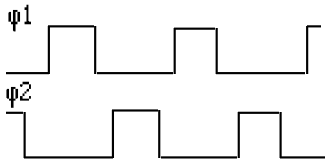
Clock Period > slowest propagation through comb. logic (rising edge to rising edge)

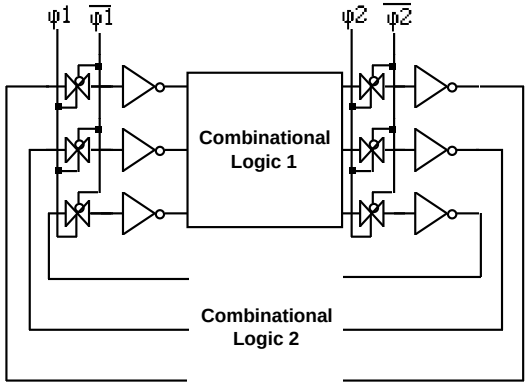
Timing Methodologies

Two Phase Non-Overlapped Clocking

Clock Waveforms:
must never overlap!

only worry about slow signals





Embedding CMOS storage element into Clocked Sequential Logic

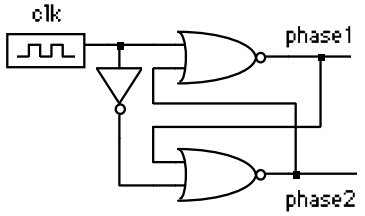
Note that Combinational Logic can be partitioned into two pieces

C/L1: inputs latched and stable by end of phase 1; compute between phases, latch outputs by end of phase 2

C/L2: just the reverse

Timing Methodologies

Generating Two-Phase Non-Overlapping Clocks

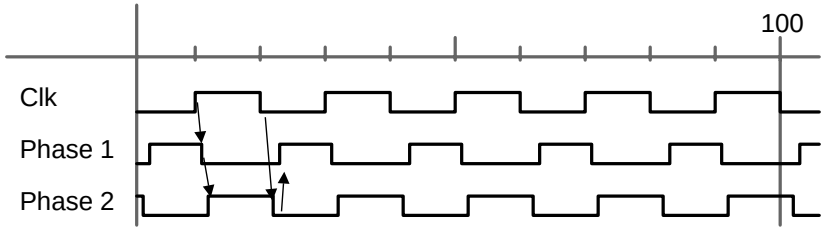


Single reference clock (or crystal)

Phase 1 high while clock is low

Phase 2 high while clock is high

Phase X cannot go high until phase Y goes low!



Non-overlap time can be increased by increasing the delay on the feedback path

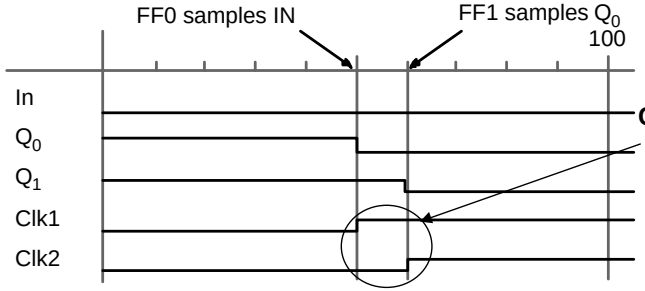
Timing Methodologies

The Problem of Clock Skew

Correct behavior assumes next state of all storage elements determined by all storage elements *at the same time*

Not possible in real systems!
" logical clock driven from more than one physical circuit with timing behavior
" different wire delay to different points in the circuit

Effect of Skew on Cascaded Flipflops:



CLK2 is a delayed version of CLK1

Original State: Q0 = 1, Q1 = 1, In = 0
Because of skew, next state becomes: Q0 = 0, Q1 = 0, not Q0 = 0, Q1 = 1

Timing Methodologies

Design Strategies for Minimizing Clock Skew

Typical propagation delays for LS FFs: 13 ns

Need substantial clock delay (on the order of 13 ns) for skew to be a problem in this relatively slow technology

Nevertheless, the following are good design practices:

" distribute clock signals in general direction of data flows

" wire carrying the clock between two communicating components should be as short as possible

" for multiphase clocked systems, distribute all clocks in similar wire paths; this minimizes the possibility of overlap

" for the non-overlap clock generate, use the phase feedback signals from the furthest point in the circuit to which the clock is distributed; this guarantees that the phase is seen as low everywhere before it allows the next phase to go high

Realing Circuits with Different Kinds of FFs

Choosing a Flipflop

R-S Clocked Latch:
- used as storage element in narrow width clocked systems
- its use is not recommended!
- however, fundamental building block of other flipflop types

J-K Flipflop:
- versatile building block
- can be used to implement D and T FFs
- usually requires least amount of logic to implement f(In,Q,Q+)
- but has two inputs with increased wiring complexity
- because of 1's catching, never use master/slave J-K FFs
- edge-triggered varieties exist

D Flipflop:
- minimizes wires, much preferred in VLSI technologies
- simplest design technique
- best choice for storage registers

T Flipflops:
- don't really exist, constructed from J-K FFs
- usually best choice for implementing counters

Preset and Clear inputs highly desirable!!

Realizing Circuits with Different Kinds of Flipflops

Characteristic Equations

R-S: $Q+ = S + \overline{R}\,Q$

D: $Q+ = D$

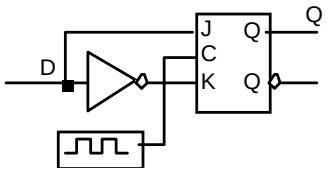
J-K: $Q+ = J\,\overline{Q} + \overline{K}\,Q$

T: $Q+ = T\,\overline{Q} + \overline{T}\,Q$

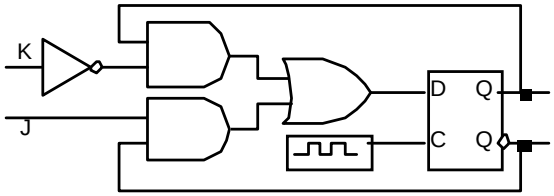
Derived from the K-maps for $Q+ = f$(Inputs, Q)

E.g., J=K=0, then Q+ = Q
J=1, K=0, then Q+ = 1
J=0, K=1, then Q+ = 0
J=1, K=1, then Q+ = Q

Implementing One FF in Terms of Another



D implemented with J-K



J-K implemented with D

Realizing Circuits with Different Kinds of Flipflops

Design Procedure

Excitation Tables: What are the necessary inputs to cause a particular kind of change in state?

Q	Q^+	R	S	J	K	T	D
0	0	X	0	0	X	0	0
0	1	0	1	1	X	1	1
1	0	1	0	X	1	1	0
1	1	0	X	X	0	0	1

Implementing D FF with a J-K FF:

1) Start with K-map of Q+ = f(D, Q)

2) Create K-maps for J and K with same inputs (D, Q)

Q \ D	0	1
0	0	1
1	0	1

$Q^+ = D$

3) Fill in K-maps with appropriate values for J and K to cause the same state changes as in the original K-map

E.g., D = Q= 0, Q+ = 0 then J = 0, K = X

Q \ D	0	1
0	0	1
1	X	X

J= D

Q \ D	0	1
0	X	X
1	1	0

K = $\overline{D}$

Realizing Circuits with Different Kinds of Flipflops

Design Procedure (Continued)

Implementing J-K FF with a D FF:

1) K-Map of Q+ = F(J, K, Q)

2,3) Revised K-map using D's excitation table its the same! that is why design procedure with D FF is simple!

Q \ JK	00	01	11	10
0	0	0	1	1
1	1	0	0	1

J (11, 10) K (00, 01)

$$Q^+ = D = J\overline{Q} + \overline{K}Q$$

Resulting equation is the combinational logic input to D to cause same behavior as J-K FF. Of course it is identical to the characteristic equation for a J-K FF.

Metastability and Asynchronous Inputs

Terms and Definitions

Clocked synchronous circuits
" common reference signal called the clock
" state of the circuit changes in relation to this clock signal

Asynchronous circuits
" inputs, state, and outputs sampled or changed independent of a common reference signal

R-S latch is asynchronous, J-K master/slave FF is synchronous

Synchronous inputs
" active only when the clock edge or level is active

Asynchronous inputs
" take effect immediately, without consideration of the clock

Compare R, S inputs of clocked transparent latch vs. plain latch

Metastability and Asynchronous Inputs

Asynchronous Inputs Are Dangerous!

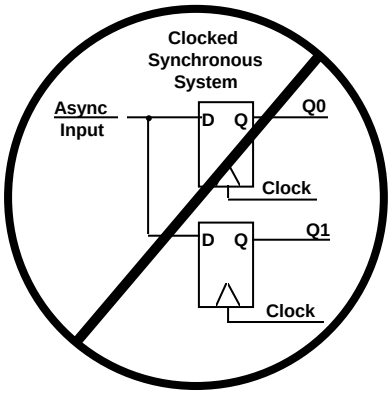
Since they take effect immediately, glitches can be disastrous

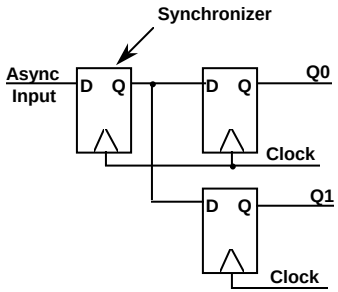
Synchronous inputs are greatly preferred!

But sometimes, asynchronous inputs cannot be avoided
e.g., reset signal, memory wait signal

Metastability and Asynchronous Outputs

Handling Asynchronous Inputs

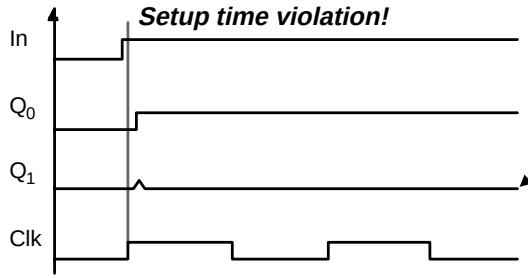




Never allow asynchronous inputs to be fanned out to more than one FF within the synchronous system

Metastability and Asynchronous Inputs

What Can Go Wrong



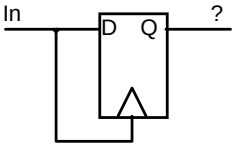
In is asynchronous
Fans out to D0 and D1
One FF catches the signal, one does not

impossible state might be reached!

Single FF that receives the asynchronous signal is *a synchronizer*

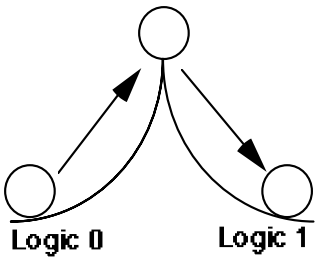
Metastability and Asynchronous Inputs

Synchronizer Failure

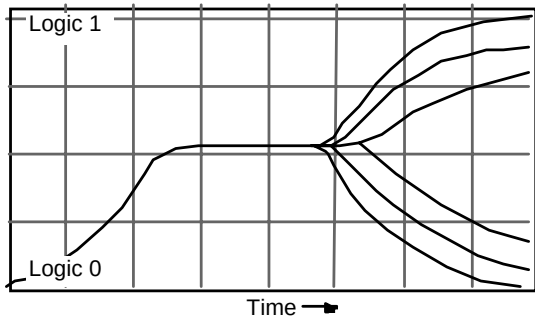


When FF input changes close to clock edge, the FF may enter the *metastable* state: neither a logic 0 nor a logic 1

It may stay in this state an indefinate amount of time, although this is not likely in real circuits



Small, but non-zero probability that the FF output will get stuck in an in-between state



Oscilloscope Traces Demonstrating Synchronizer Failure and Eventual Decay to Steady State

Metastability and Asynchronous Inputs

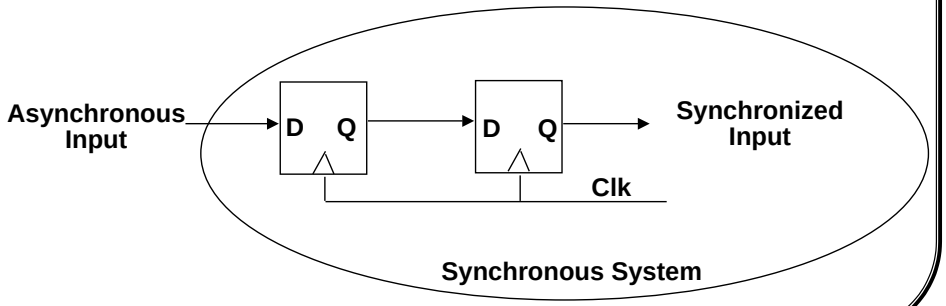
Solutions to Synchronizer Failure

" the probability of failure can never be reduced to 0, but it can be reduced

" slow down the system clock
this gives the synchronizer more time to decay into a steady state
synchronizer failure becomes a big problem for very high speed systems

" use fastest possible logic in the synchronizer
this makes for a very sharp "peak" upon which to balance
S or AS TTL D-FFs are recommended

" cascade two synchronizers



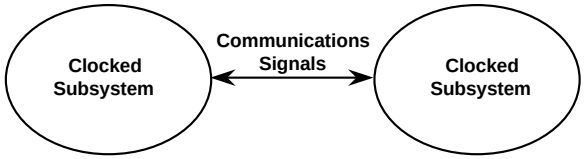
Self-Timed and Speed Independent Circuits

Limits of Synchronous Systems

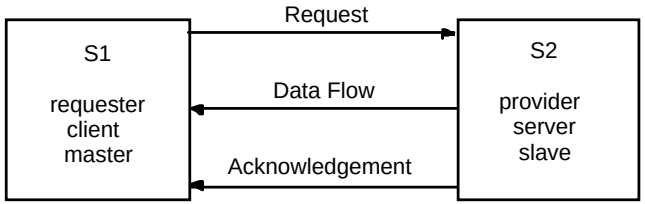
Fully synchronous not possible for very large systems because of problems of clock skew

Partition system into components that are locally clocked

These communicate using "speed independent" protocols

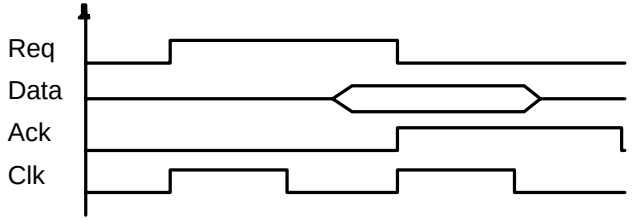


Request/Acknowledgement Signaling

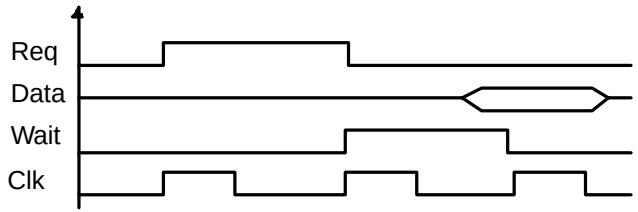


Self-Timed and Speed Independent Circuits

Synchronous Signaling



Master issues read request; Slave produces data and acks back



Alternative Synchronous Scheme:

Slave issues WAIT signal if it cannot satisfy request in one clock cycle

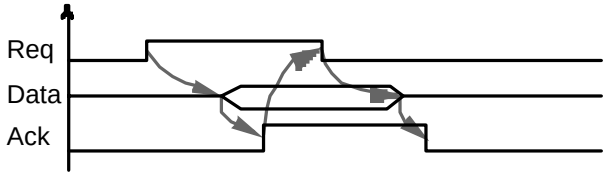
Self-Timed and Speed Independent Circuits

Asynchronous/Speed Independent Signaling

Communicate information by signal levels rather than edges!

No clock signal

4 Cycle Signaling/Return to Zero Signaling



(1) master raises request
slave performs request

(2) slave "done" by raising acknowledge

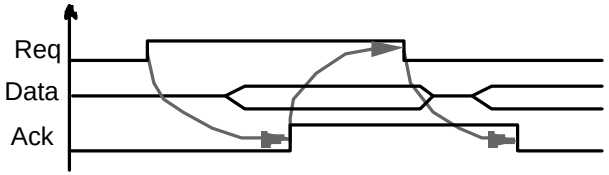
(3) master latches data
acks by lowering request

(4) slave resets self by lowing acknowledge signal

Self-Timed and Speed Independent Circuits

Alternative: 2 cycle signaling

Non-Return-to-Zero



(1) master raises request
slave services request

(2) slave indicates that it is done by raising acknowledge

Next request indicated by low level of request

Requires additional state in master and slave to remember previous setting or request/acknowledge

4 Cycle Signaling is more foolproof

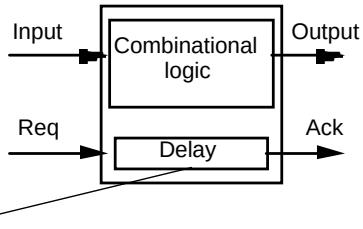
Self-Timed and Speed Independent Circuits

Self-Timed Circuits

Determine on their own when a given request has been serviced

No internal clocks

Usually accomplished by modeling worse case delay within self-timed component



Models worst case delay
e.g., if combinational logic is 5 gate levels deep, delay line between request in and ack out is also 5 levels deep

Chapter Summary

" Fundamental Building Block of Circuits with State: latch and flipflop

" R-S Latch, J-K master/slave Flipflop, Edge-triggered D Flipflop

" Clocking Methodologies:

For latches: Narrow width clocking vs. Multiphase Non-overlapped
Narrow width clocking and two sided timing constraints
Two phase clocking and single sided timing constraints

For FFs: Single phase clocking with edge triggered flipflops

Cascaded FFs work because propagation delays exceed hold times

Beware of Clock Skew

" Asynchronous Inputs and Their Dangers

Synchronizer Failure: What it is and how to minimize its impact

" Speed Independent Circuits

Asynchronous Signaling Conventions: 4 and 2 Cycle Handshakes

Self-Timed Circuits

Chapter #7: Sequential Logic Case Studies

Contemporary Logic Design

Randy H. Katz
University of California, Berkeley

June 1993

Motivation

- ***Flipflops:*** **most primitive "packaged" sequential circuits**
- ***More complex sequential building blocks:***

  Storage registers, Shift registers, Counters
  Available as components in the TTL Catalog

- ***How to represent and design simple sequential circuits:*** **counters**
- ***Problems and pitfalls when working with counters:***

  Start-up States
  Asynchronous vs. Synchronous logic

Chapter Overview

Examine Real Sequential Logic Circuits Available as Components

- **Registers for storage and shifting**
- **Random Access Memories**
- **Counters**

Counter Design Procedure

- **Simple but useful finite state machine**
- **State Diagram, State Transition Table, Next State Functions**
- **Excitation Tables for implementation with alternative flipflop types**

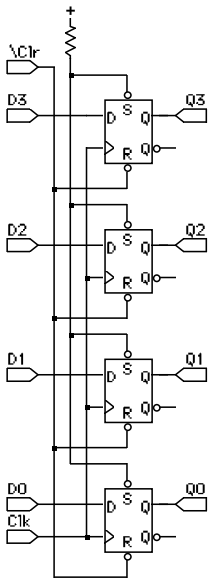
Synchronous vs. Asynchronous Counters

- **Ripple vs. Synchronous Counters**
- **Asynchronous vs. Synchronous Clears and Loads**

Kinds of Registers and Counters

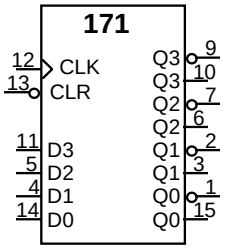
Storage Register

Group of storage elements read/written as a unit



4-bit register constructed from 4 D FFs
Shared clock and clear lines

Schematic Shape

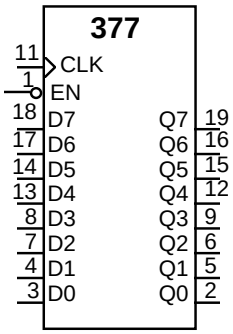


TTL 74171 Quad D-type FF with Clear
(Small numbers represent pin #s on package)

Kinds of Registers and Counters

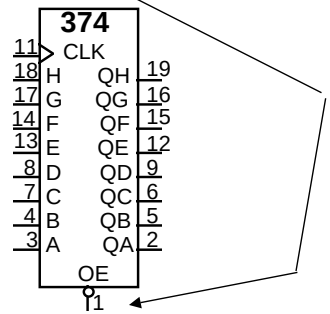
Input/Output Variations

Selective Load Capability
Tri-state or Open Collector Outputs
True and Complementary Outputs



74377 Octal D-type FFs with input enable

EN enabled low and lo-to-hi clock transition to load new data into register



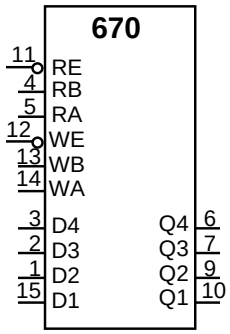
74374 Octal D-type FFs with output enable

OE asserted low presents FF state to output pins; otherwise high impedence

Kinds of Registers and Counters

Register Files

Two dimensional array of flipflops
Address used as index to a particular word
Word contents read or written



74670 4x4 Register File with Tri-state Outputs

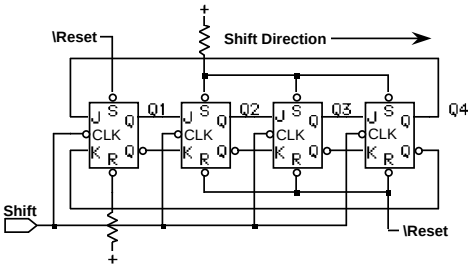
Separate Read and Write Enables
Separate Read and Write Address
Data Input, Q Outputs

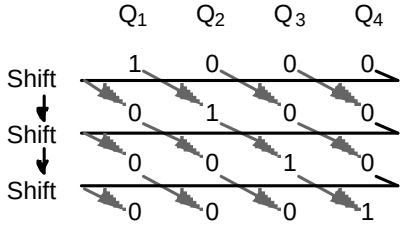
Contains 16 D-ffs, organized as four rows (words) of four elements (bits)

Kinds of Registers and Counters

Shift Registers

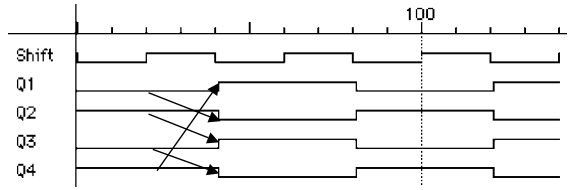
Storage + ability to circulate data among storage elements





Shift from left storage element to right neighbor on every lo-to-hi transition on shift signal

Wrap around from rightmost element to leftmost element

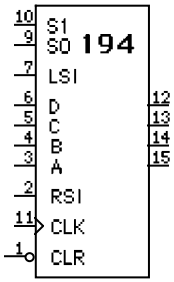


Master Slave FFs: sample inputs while clock is high; change outputs on falling edge

Kinds of Registers and Counters

Shift Register I/O

Serial vs. Parallel Inputs
Serial vs. Parallel Outputs
Shift Direction: Left vs. Right



74194 4-bit Universal Shift Register

Serial Inputs: LSI, RSI
Parallel Inputs: D, C, B, A
Parallel Outputs: QD, QC, QB, QA
Clear Signal
Positive Edge Triggered Devices

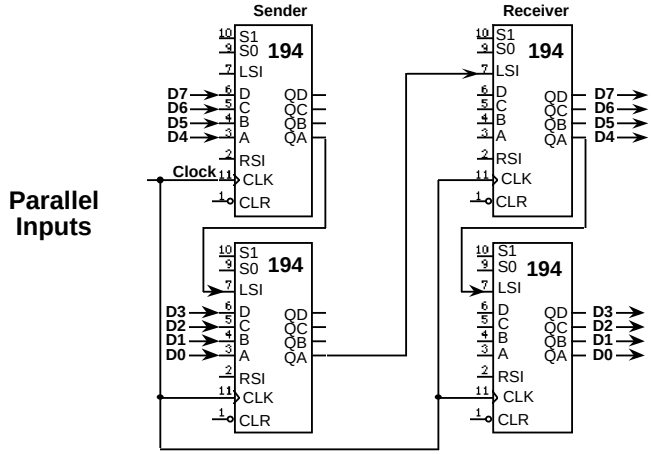
S1,S0 determine the shift function
S1 = 1, S0 = 1: Load on rising clk edge
synchronous load
S1 = 1, S0 = 0: shift left on rising clk edge
LSI replaces element D
S1 = 0, S0 = 1: shift right on rising clk edge
RSI replaces element A
S1 = 0, S0 = 0: hold state

Multiplexing logic on input to each FF!

Shifters well suited for serial-to-parallel conversions, such as terminal to computer communications

Kinds of Registers and Counters

Shift Register Application: Parallel to Serial Conversion



Parallel Inputs

Parallel Outputs

Serial transmission

Kinds of Registers and Counters

Counters

Proceed through a well-defined sequence of states in response to count signal

3 Bit Up-counter: 000, 001, 010, 011, 100, 101, 110, 111, 000, ...

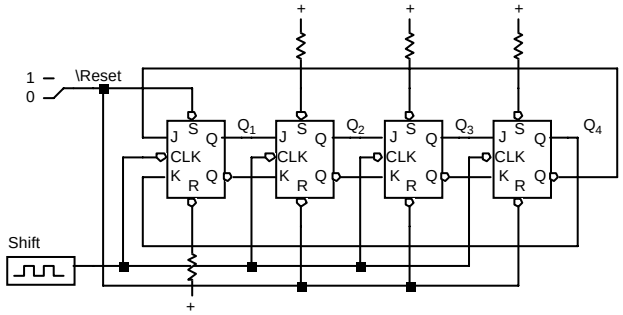
3 Bit Down-counter: 111, 110, 101, 100, 011, 010, 001, 000, 111, ...

Binary vs. BCD vs. Gray Code Counters

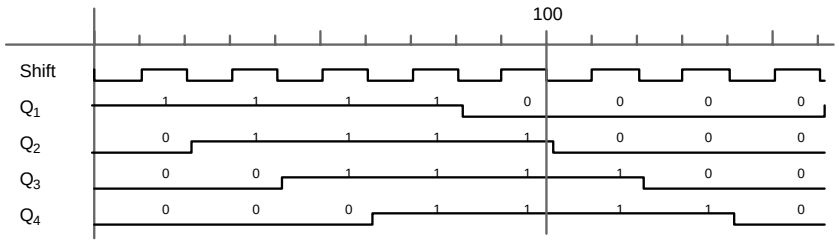
A counter is a "degenerate" finite state machine/sequential circuit where the state *is* the only output

Kinds of Registers and Counters

Johnson (Mobius) Counter



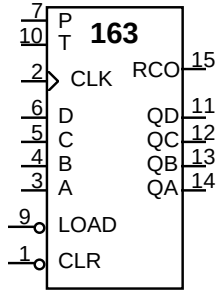
End-Around



8 possible states, single bit change per state, useful for avoiding glitches

Kinds of Registers and Counters

Catalog Counter



74163 Synchronous 4-Bit Upcounter

Synchronous Load and Clear Inputs

Positive Edge Triggered FFs

Parallel Load Data from D, C, B, A

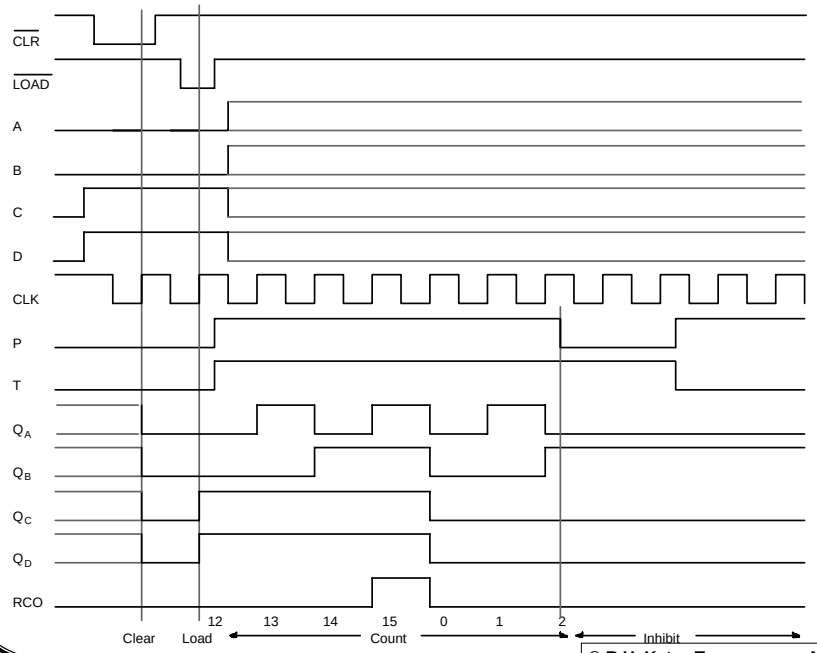
P, T Enable Inputs: both must be asserted to enable counting

RCO: asserted when counter enters its highest state 1111, used for cascading counters *"Ripple Carry Output"*

74161: similar in function, asynchronous load and reset

Kinds of Registers and Counters

74163 Detailed Timing Diagram



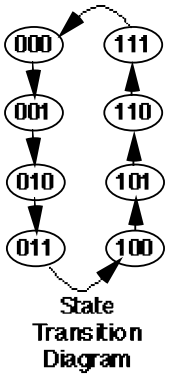
Counter Design Procedure

Introduction

This procedure can be generalized to implement ANY finite state machine

Counters are a very simple way to start: no decisions on what state to advance to next current state is the output

Example: **3-bit Binary Upcounter**



State Transition Diagram

Present State			Next State			Flipflop Inputs		
C	B	A	C+	B+	A+	TC	TB	TA
0	0	0	0	0	1			
0	0	1	0	1	0			
0	1	0	0	1	1			
0	1	1	1	0	0			
1	0	0	1	0	1			
1	0	1	1	1	0			
1	1	0	1	1	1			
1	1	1	0	0	0			

State Transition Table

Flipflop Input Table

Decide to implement with Toggle Flipflops

What inputs must be presented to the T FFs to get them to change to the desired state bit?

This is called "Remapping the Next State Function"

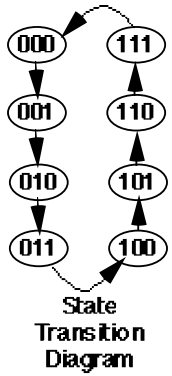
Counter Design Procedure

Introduction

This procedure can be generalized to implement ANY finite state machine

Counters are a very simple way to start: no decisions on what state to advance to next current state is the output

Example: **3-bit Binary Upcounter**



State Transition Diagram

Present State			Next State			Flipflop Inputs		
C	B	A	C+	B+	A+	TC	TB	TA
0	0	0	0	0	1	0	0	1
0	0	1	0	1	0	0	1	1
0	1	0	0	1	1	0	0	1
0	1	1	1	0	0	1	1	1
1	0	0	1	0	1	0	0	1
1	0	1	1	1	0	0	1	1
1	1	0	1	1	1	0	0	1
1	1	1	0	0	0	1	1	1

State Transition Table

Flipflop Input Table

Decide to implement with Toggle Flipflops

What inputs must be presented to the T FFs to get them to change to the desired state bit?

This is called "Remapping the Next State Function"

Counter Design Procedure

Example Continued

K-maps for Toggle Inputs:

Resulting Logic Circuit:

A \ CB	00	01	11	10
0				
1				

TA =

A \ CB	00	01	11	10
0				
1				

TB =

A \ CB	00	01	11	10
0				
1				

TC =

Counter Design Procedure

Example Continued

K-maps for Toggle Inputs:

A \ CB	00	01	11	10
0	1	1	1	1
1	1	1	1	1

TA = 1

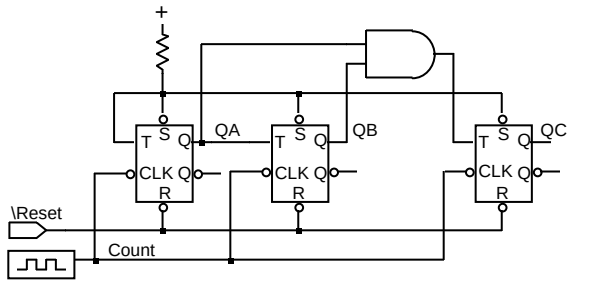
A \ CB	00	01	11	10
0	0	0	0	0
1	1	1	1	1

TB = A

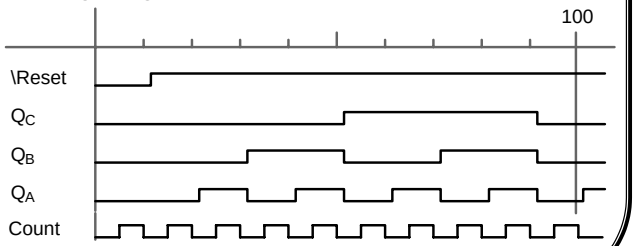
A \ CB	00	01	11	10
0	0	0	0	0
1	0	1	1	0

TC = A• B

Resulting Logic Circuit:



Timing Diagram:

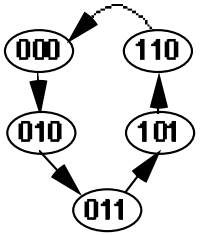


Counter Design Procedure

More Complex Count Sequence

Step 1: **Derive the State Transition Diagram**

Count sequence: 000, 010, 011, 101, 110



Present State			Next State		
C	B	A	C+	B+	A+
0	0	0			
0	0	1			
0	1	0			
0	1	1			
1	0	0			
1	0	1			
1	1	0			
1	1	1			

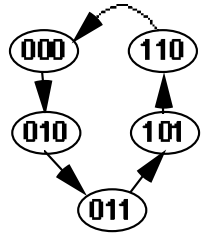
Step 2: **State Transition Table**

Counter Design Procedure

More Complex Count Sequence

Step 1: **Derive the State Transition Diagram**

Count sequence: 000, 010, 011, 101, 110



Present State			Next State		
C	B	A	C+	B+	A+
0	0	0	0	1	0
0	0	1	X	X	X
0	1	0	0	1	1
0	1	1	1	0	1
1	0	0	X	X	X
1	0	1	1	1	0
1	1	0	0	0	0
1	1	1	X	X	X

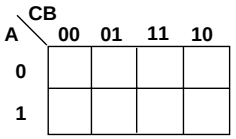
Step 2: **State Transition Table**

Note the Don't Care conditions

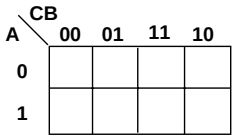
Counter Design Procedure

More Complex Count Sequence

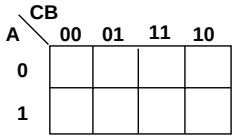
Step 3: **K-Maps for Next State Functions**



C+ =



B+ =

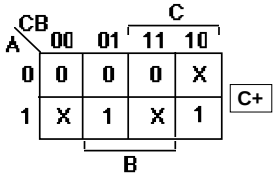


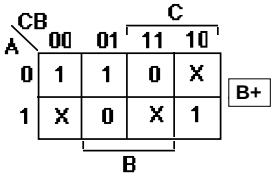
A+ =

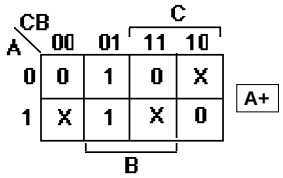
Counter Design Procedure

More Complex Count Sequence

Step 3: **K-Maps for Next State Functions**







Counter Design Procedure

More Complex Counter Sequencing

Step 4: **Choose Flipflop Type for Implementation**
Use Excitation Table to Remap Next State Functions

Q	Q+	T
0	0	0
0	1	1
1	0	1
1	1	0

Toggle Excitation Table

Present State			Toggle Inputs		
C	B	A	TC	TB	TA
0	0	0			
0	0	1			
0	1	0			
0	1	1			
1	0	0			
1	0	1			
1	1	0			
1	1	1			

Remapped Next State Functions

Counter Design Procedure

More Complex Counter Sequencing

Step 4: **Choose Flipflop Type for Implementation**
Use Excitation Table to Remap Next State Functions

Q	Q+	T
0	0	0
0	1	1
1	0	1
1	1	0

Toggle Excitation Table

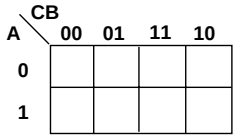
Present State			Toggle Inputs		
C	B	A	TC	TB	TA
0	0	0	0	1	0
0	0	1	X	X	X
0	1	0	0	0	1
0	1	1	1	1	0
1	0	0	X	X	X
1	0	1	0	1	1
1	1	0	1	1	0
1	1	1	X	X	X

Remapped Next State Functions

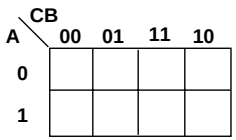
Counter Design Procedure

More Complex CounterSequencing

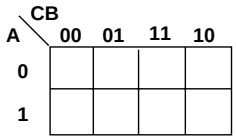
Remapped K-Maps



TC



TB



TA

***TC* =**

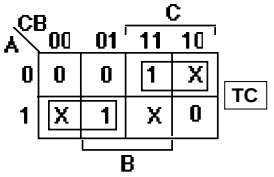
***TB* =**

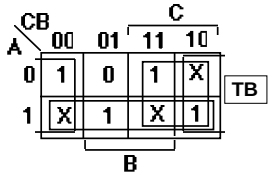
***TA* =**

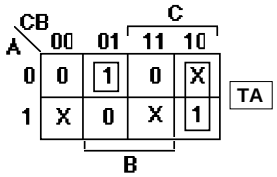
Counter Design Procedure

More Complex CounterSequencing

Remapped K-Maps







***TC* = $\bar{A}$ C + A $\bar{C}$ = A xor C**

***TB* = A + $\bar{B}$ + C**

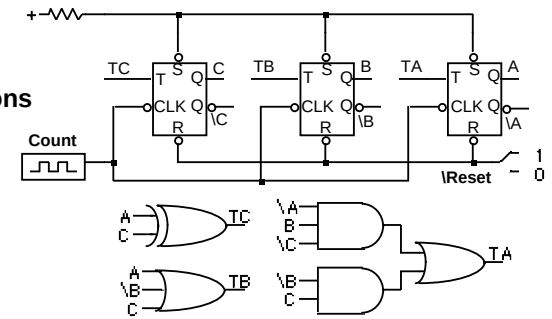
***TA* = $\bar{A}$ B $\bar{C}$ + $\bar{B}$ C**

Counter Design Procedure

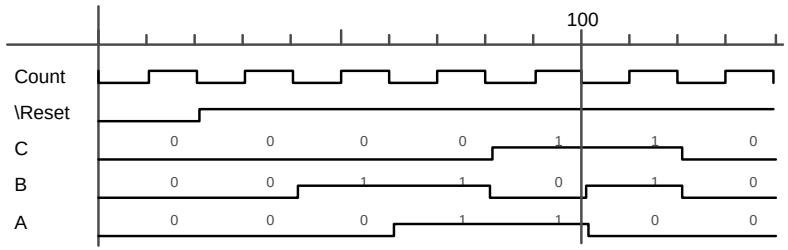
More Complex Counter Sequencing

Resulting Logic:

5 Gates
13 Input Literals +
Flipflop connections



Timing Waveform:



Self-Starting Counters

Start-Up States

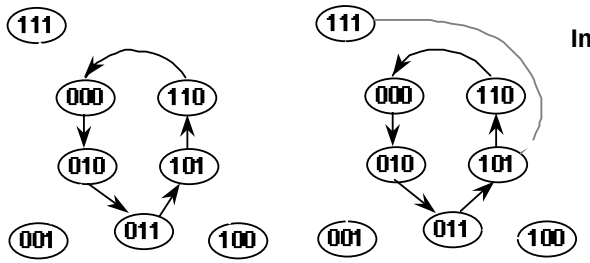
At power-up, counter may be in possible state

Designer must guarantee that it (eventually) enters a valid state

Especially a problem for counters that validly use a subset of states

Self-Starting Solution:

Design counter so that even the invalid states eventually transition to valid state



Implementation in Previous Slide!

Two Self-Starting State Transition Diagrams for the Example Counter

Self-Starting Counters

Deriving State Transition Table from Don't Care Assignment

Inputs to Toggle Flip-flops

A \ CB	00	01	11	10
0	0	0	1	1
1	1	1	0	0

TC

A \ CB	00	01	11	10
0	1	0	1	1
1	1	1	1	1

TB

A \ CB	00	01	11	10
0	0	1	0	1
1	0	0	0	1

TA

State Changes

A \ CB	00	01	11	10
0	0	0	0	0
1	1	1	1	1

C+

A \ CB	00	01	11	10
0	1	1	0	1
1	1	0	0	1

B+

A \ CB	00	01	11	10
0	0	1	0	1
1	1	1	1	0

A+

State Transition Table

Present State C	B	A	Next State C+	B+	A+
0	0	0	0	1	0
0	0	1	1	1	1
0	1	0	0	1	1
0	1	1	1	0	1
1	0	0	0	1	1
1	0	1	1	1	0
1	1	0	0	0	0
1	1	1	1	0	1

Implementation with Different Kinds of FFs

R-S Flipflops

Continuing with the 000, 010, 011, 101, 110, 000, ... counter example

Q	Q+	R	S
0	0	X	0
0	1	0	1
1	0	1	0
1	1	0	X

$Q+ = S + R\,\bar{Q}$

RS Exitation Table

Present State C	B	A	Next State C+	B+	A+	Remapped Next State RC	SC	RB	SB	RA	SA
0	0	0	0	1	0						
0	0	1	X	X	X						
0	1	0	0	1	1						
0	1	1	1	0	1						
1	0	0	X	X	X						
1	0	1	1	1	0						
1	1	0	0	0	0						
1	1	1	X	X	X						

Remapped Next State Functions

Implementation with Different Kinds of FFs

R-S Flipflops

Continuing with the 000, 010, 011, 101, 110, 000, ... counter example

Q	Q+	R	S
0	0	X	0
0	1	0	1
1	0	1	0
1	1	0	X

$Q+ = S + R\,\bar{Q}$

RS Exitation Table

Present State C	B	A	Next State C+	B+	A+	Remapped Next State RC	SC	RB	SB	RA	SA
0	0	0	0	1	0	X	0	0	1	X	0
0	0	1	X	X	X	X	X	X	X	X	X
0	1	0	0	1	1	X	0	0	X	0	1
0	1	1	1	0	1	0	1	1	0	0	X
1	0	0	X	X	X	X	X	X	X	X	X
1	0	1	1	1	0	0	X	0	1	1	0
1	1	0	0	0	0	1	0	1	0	X	0
1	1	1	X	X	X	X	X	X	X	X	X

Remapped Next State Functions

Implementation with Different Kinds of FFs

RS FFs Continued

A \ CB	00	01	11	10
0				
1				

RC

A \ CB	00	01	11	10
0				
1				

SC

A \ CB	00	01	11	10
0				
1				

RB

A \ CB	00	01	11	10
0				
1				

SB

A \ CB	00	01	11	10
0				
1				

RA

A \ CB	00	01	11	10
0				
1				

SA

RC =

SC =

RB =

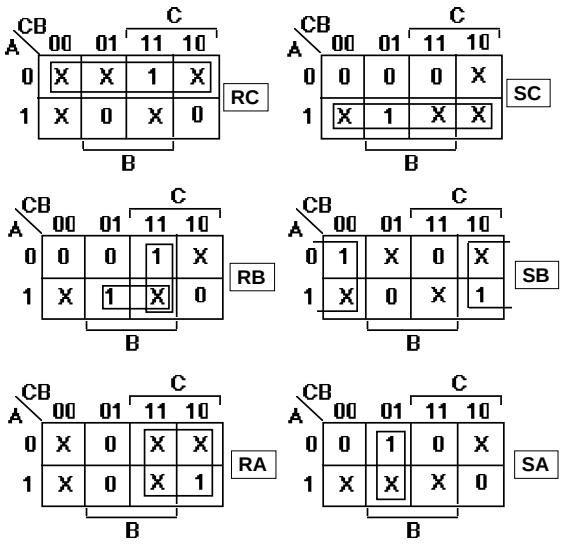
SB =

RA =

SA =

Implementation with Different Kinds of FFs

RS FFs Continued



RC = $\overline{A}$

SC = A

RB = A B + B C

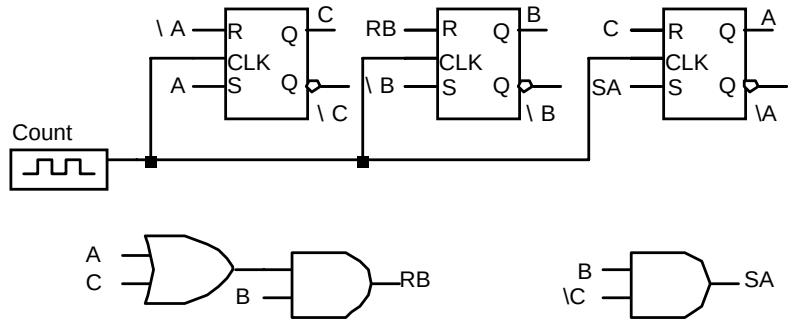
SB = $\overline{B}$

RA = C

SA = B $\overline{C}$

Implementation With Different Kinds of FFs

RS FFs Continued



Resulting Logic Level Implementation:
3 Gates, 11 Input Literals + Flipflop connections

Implementation with Different FF Types

J-K FFs

Q	Q+	J	K
0	0	0	X
0	1	1	X
1	0	X	1
1	1	X	0

Q+ = J $\overline{Q}$ + $\overline{K}$ Q

J-K Excitation Table

Present State			Next State			Remapped Next State					
C	B	A	C+	B+	A+	JC	KC	JB	KB	JA	KA
0	0	0	0	1	0						
0	0	1	X	X	X						
0	1	0	0	1	1						
0	1	1	1	0	1						
1	0	0	X	X	X						
1	0	1	1	1	0						
1	1	0	0	0	0						
1	1	1	X	X	X						

Remapped Next State Functions

Implementation with Different FF Types

J-K FFs

Q	Q+	J	K
0	0	0	X
0	1	1	X
1	0	X	1
1	1	X	0

Q+ = J $\overline{Q}$ + $\overline{K}$ Q

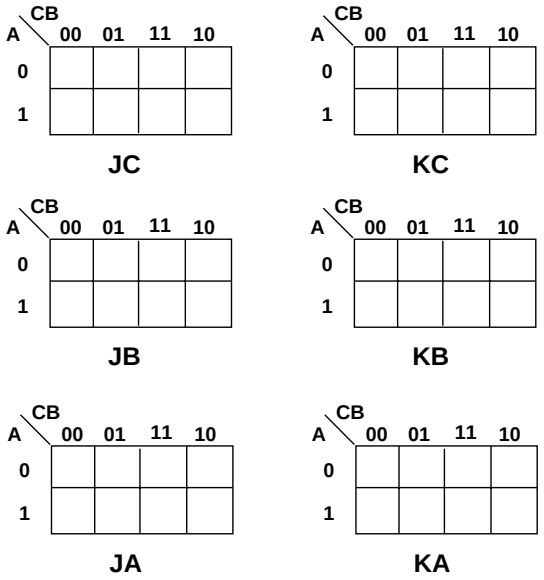
J-K Excitation Table

Present State			Next State			Remapped Next State					
C	B	A	C+	B+	A+	JC	KC	JB	KB	JA	KA
0	0	0	0	1	0	0	X	1	X	0	X
0	0	1	X	X	X	X	X	X	X	X	X
0	1	0	0	1	1	0	X	X	0	1	X
0	1	1	1	0	1	1	X	X	1	X	0
1	0	0	X	X	X	X	X	X	X	X	X
1	0	1	1	1	0	X	0	1	X	X	1
1	1	0	0	0	0	X	1	X	1	0	X
1	1	1	X	X	X	X	X	X	X	X	X

Remapped Next State Functions

Implementation with Different FF Types

J-K FFs Continued



***JC* =**

***KC* =**

***JB* =**

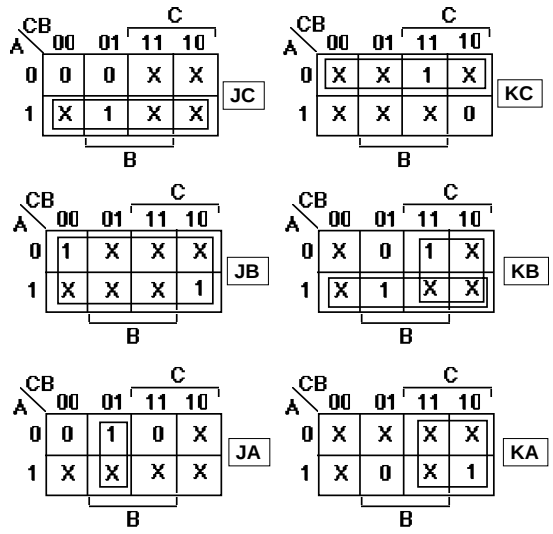
***KB* =**

***JA* =**

***KA* =**

Implementation with Different FF Types

J-K FFs Continued



***JC* = A**

***KC* = $\overline{A}$**

***JB* = 1**

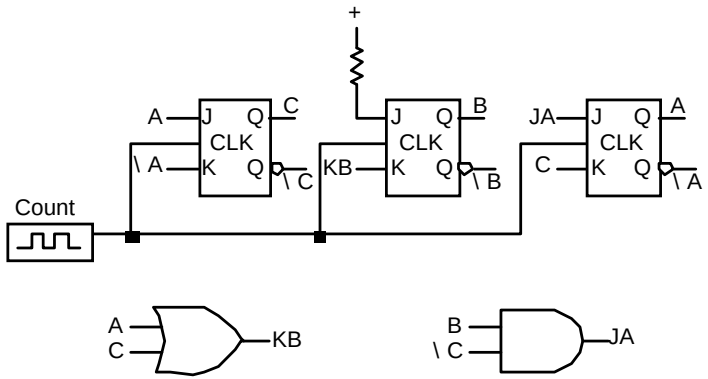
***KB* = A + C**

***JA* = B $\overline{C}$**

***KA* = C**

Implementation with Different FF Types

J-K FFs Continued



Resulting Logic Level Implementation:
2 Gates, 10 Input Literals + Flipflop Connections

Implementation with Different FF Types

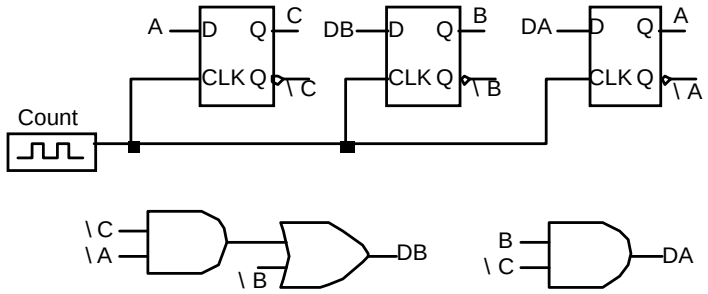
D FFs

Simplest Design Procedure: No remapping needed!

***DC* = A**

***DB* = $\overline{A}\,\overline{C}$ + $\overline{B}$**

***DA* = B $\overline{C}$**



Resulting Logic Level Implementation:
3 Gates, 8 Input Literals + Flipflop connections

Implementation with Different FF Types

Comparison

" T FFs well suited for straightforward binary counters

But yielded worst gate and literal count for this example!

" No reason to choose R-S over J-K FFs: it is a proper subset of J-K

R-S FFs don't really exist anyway

J-K FFs yielded lowest gate count

Tend to yield best choice for packaged logic where gate count is key

" D FFs yield simplest design procedure

Best literal count

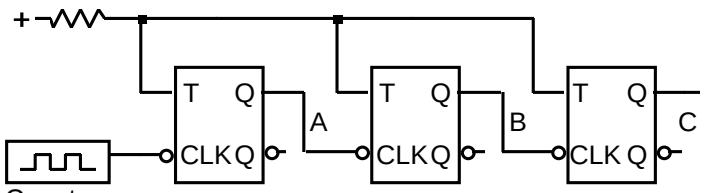
D storage devices very transistor efficient in VLSI

Best choice where area/literal count is the key

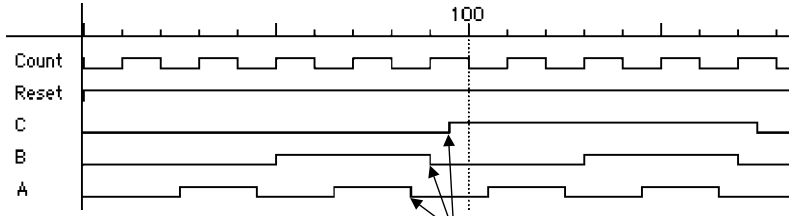
Asynchronous vs. Synchronous Counters

Ripple Counters

Deceptively attractive alternative to synchronous design style



Count signal ripples from left to right

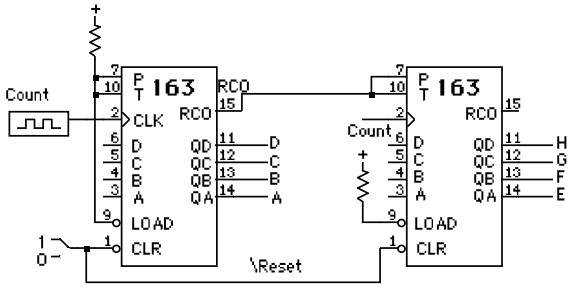


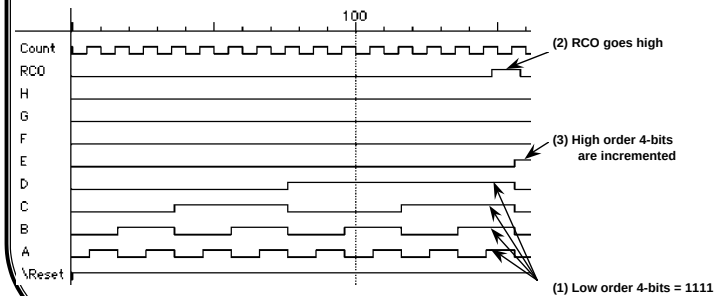
State transitions are not sharp!

Can lead to "spiked outputs" from combinational logic decoding the counter's state

Asynchronous vs. Synchronous Counters

Cascaded Synchronous Counters with Ripple Carry Outputs





First stage RCO enables second stage for counting

RCO asserted soon after stage enters state 1111

also a function of the T Enable

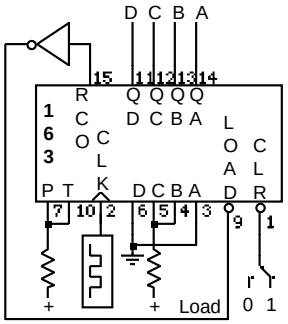
Downstream stages lag in their 1111 to 0000 transitions

Affects Count period and decoding logic

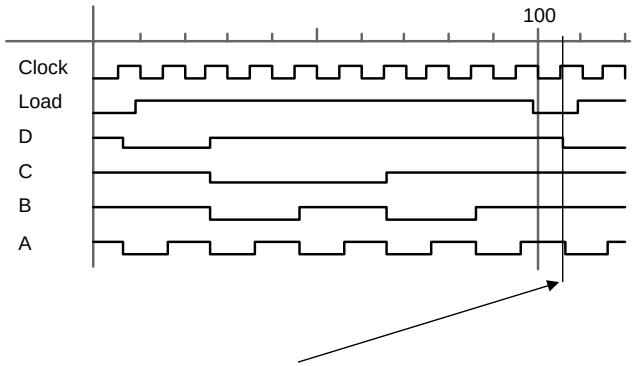
Asynchronous vs. Synchronous Counters

The Power of Synchronous Clear and Load

Starting Offset Counters:
e.g., 0110, 0111, 1000, 1001, 1010, 1011, 1100, 1101, 1111, 0110, ...



0110 is the state to be loaded



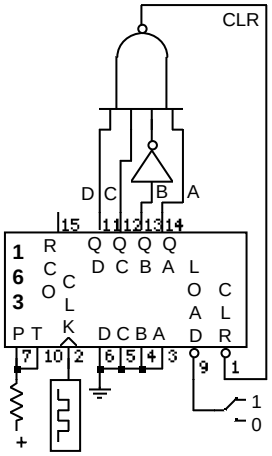
Use RCO signal to trigger Load of a new state

Since 74163 Load is synchronous, state changes only on the next rising clock edge

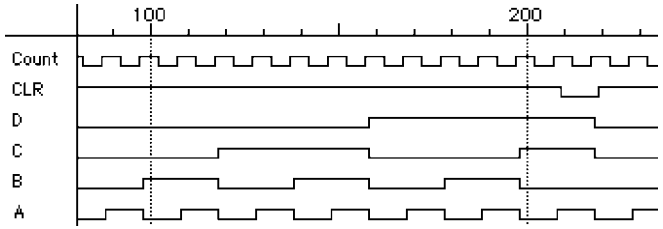
Asynchronous vs. Synchronous Counters

Offset Counters Continued

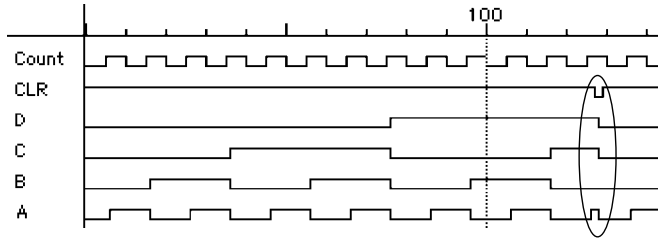
Ending Offset Counter:
e.g., 0000, 0001, 0010, ..., 1100, 1101, 0000



Decode state to determine when to reset to 0000



Clear signal takes effect on the rising count edge



Replace '163 with '161, Counter with Async Clear
Clear takes effect immediately!

Random Access Memories

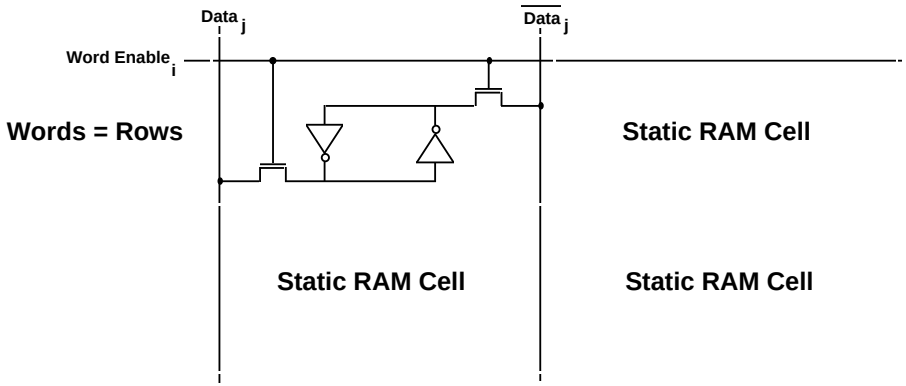
Static RAM

Transistor efficient methods for implementing storage elements

Small RAM: 256 words by 4-bit

Large RAM: 4 million words by 1-bit

We will discuss a 1024 x 4 organization



Columns = Bits (Double Rail Encoded)

Random Access Memories

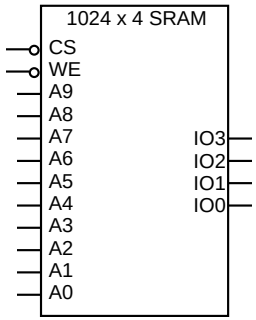
Static RAM Organization

Chip Select Line (active lo)

Write Enable Line (active lo)

10 Address Lines

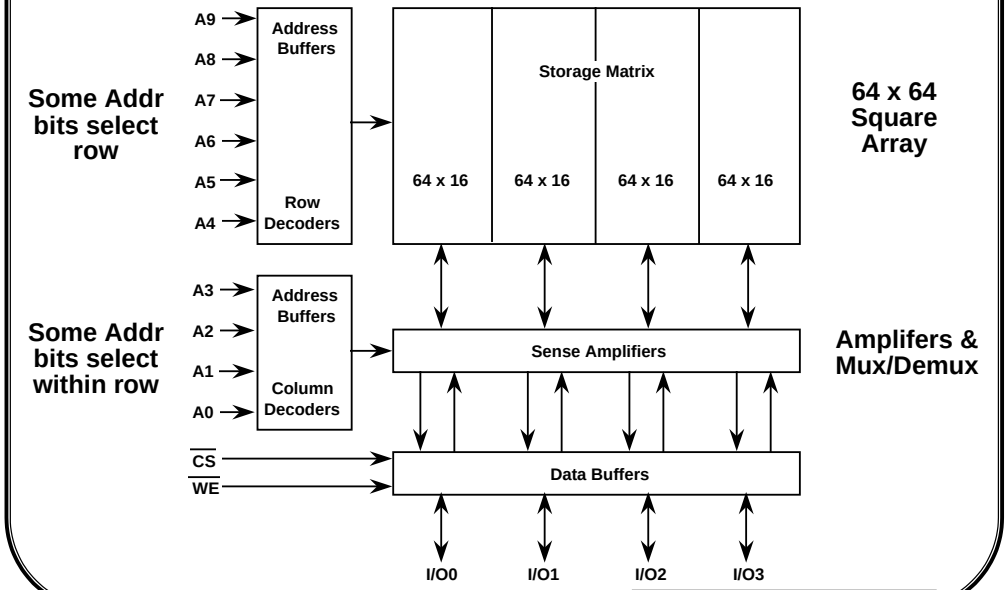
4 Bidirectional Data Lines



Random Access Memories

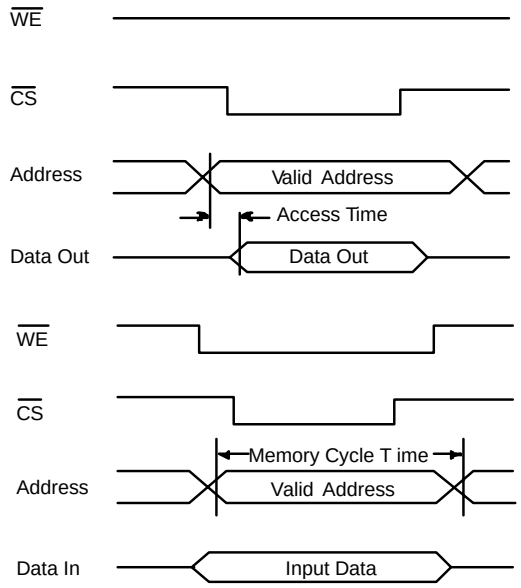
RAM Organization

Long thin layouts are not the best organization for a RAM



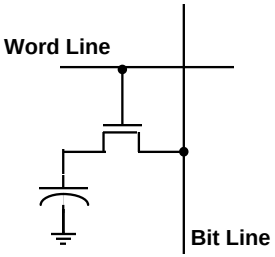
Random Access Memories

RAM Timing



Random Access Memories

Dynamic RAMs



1 Transistor (+ capacitor) memory element

Read: Assert Word Line, Sense Bit Line

Write: Drive Bit Line, Assert Word Line

Destructive Read-Out

Need for Refresh Cycles: storage decay in ms

Internal circuits read word and write back

Random Access Memories

DRAM Organization

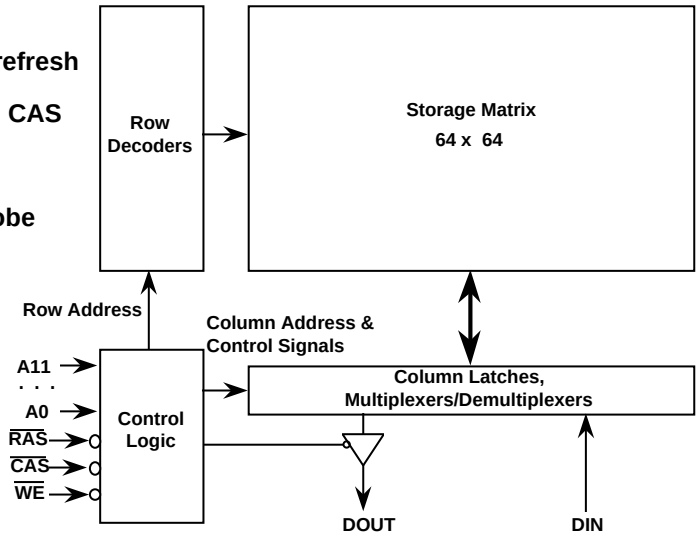
Long rows to simplify refresh

Two new signals: RAS, CAS

Row Address Strobe

Column Address Strobe

replace Chip Select

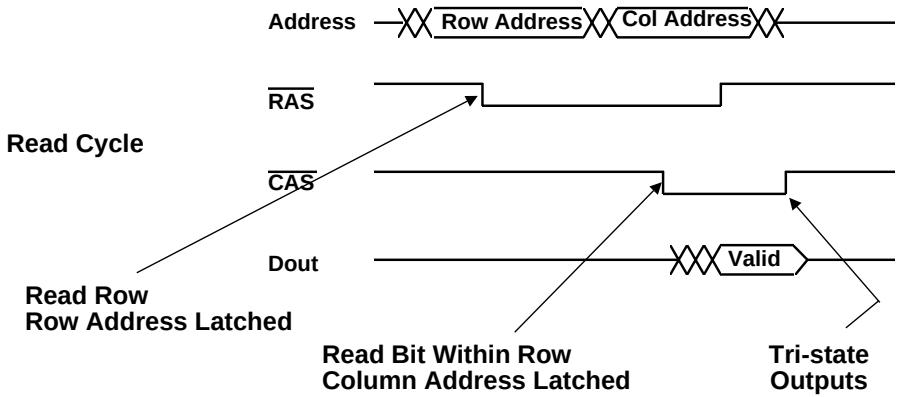


Random Access Memory

RAS, CAS Addressing

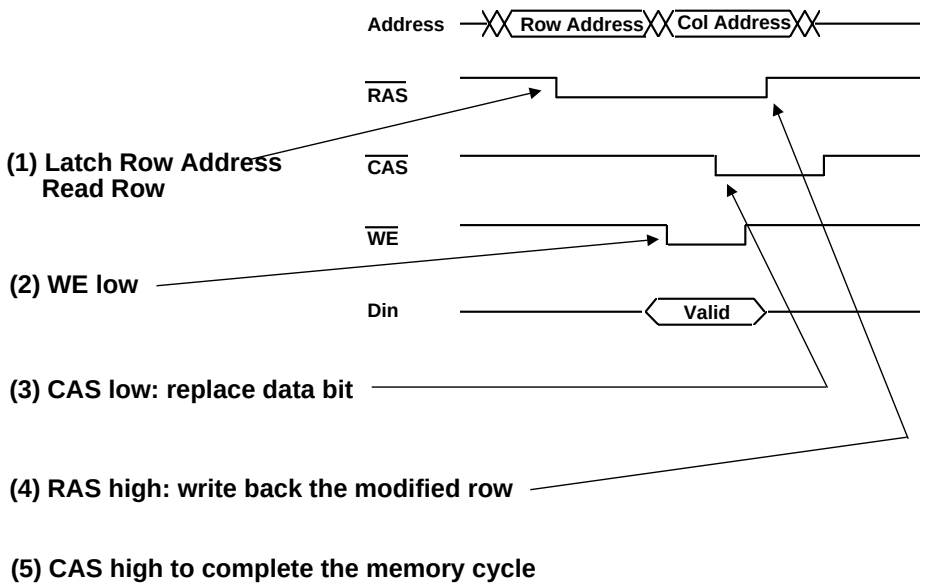
Even to read 1 bit, an entire 64-bit row is read!

Separate addressing into two cycles: Row Address, Column Address
Saves on package pins, speeds RAM access for sequential bits!



Random Access Memory

Write Cycle Timing



Random Access Memory

RAM Refresh

Refresh Frequency:

- 4096 word RAM -- refresh each word once every 4 ms
- Assume 120ns memory access cycle
- This is one refresh cycle every 976 ns (1 in 8 DRAM accesses)!
- But RAM is really organized into 64 rows
- This is one refresh cycle every 62.5 µs (1 in 500 DRAM accesses)
- Large capacity DRAMs have 256 rows, refresh once every 16 µs

RAS-only Refresh (RAS cycling, no CAS cycling)

- External controller remembers last refreshed row

Some memory chips maintain refresh row pointer

- CAS before RAS refresh: if CAS goes low before RAS, then refresh

Random Access Memory

DRAM Variations

Page Mode DRAM:

- read/write bit within last accessed row without RAS cycle
- RAS, CAS, CAS, . . ., CAS, RAS, CAS, ...
- New column address for each CAS cycle

Static Column DRAM:

- like page mode, except address bit changes signal new cycles rather than CAS cycling
- on writes, deselect chip or CAS while address lines are changing

Nibble Mode DRAM:

- like page mode, except that CAS cycling implies next column address in sequence -- no need to specify column address after first CAS
- Works for 4 bits at a time (hence "nibble")
  RAS, CAS, CAS, CAS, CAS, RAS, CAS, CAS, CAS, CAS, . . .

Chapter Summary

" The Variety of Sequential Circuit Packages
  Registers, Shifters, Counters, RAMs

" Counters as Simple Finite State Machines

" Counter Design Procedure
  1. Derive State Diagram
  2. Derive State Transition Table
  3. Determine Next State Functions
  4. Remap Next State Functions for Target FF Types
     Using Excitation Tables; Implement Logic

" Different FF Types in Counters
  J-K best for reducing gate count in packaged logic
  D is easiest design plus best for reducing wiring and area in VLSI

" Asynchronous vs. Synchronous Counters
  Avoid Ripple Counters! State transitions are not sharp
  Beware of potential problems when cascading synchronous counters

  Offset counters: easy to design with synchronous load and clear
  Never use counters with asynchronous clear for this kind of application

Chapter #8: Finite State Machine Design

Contemporary Logic Design

Randy H. Katz
University of California, Berkeley

June 1993

Motivation

- ***Counters:*** **Sequential Circuits where State = Output**
- ***Generalizes to Finite State Machines:***
  Outputs are Function of State (and Inputs)
  Next States are Functions of State and Inputs
  Used to implement circuits that control other circuits
  "Decision Making" logic
- ***Application of Sequential Logic Design Techniques***
  Word Problems
  Mapping into formal representations of FSM behavior
  Case Studies

Chapter Overview

Concept of the State Machine

- **Partitioning into Datapath and Control**
- **When Inputs are Sampled and Outputs Asserted**

Basic Design Approach

- **Six Step Design Process**

Alternative State Machine Representations

- **State Diagram, ASM Notation, VHDL, ABEL Description Language**

Moore and Mealy Machines

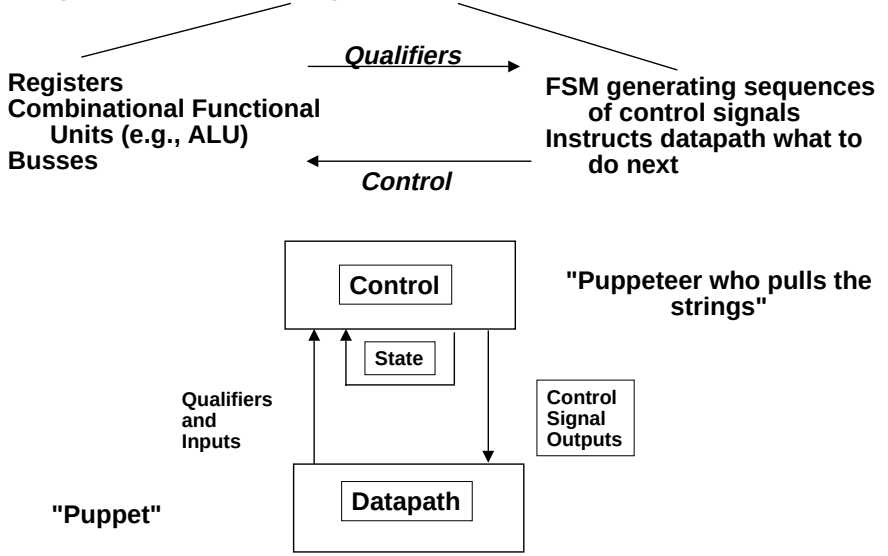
- **Definitions, Implementation Examples**

Word Problems

- **Case Studies**

Concept of the State Machine

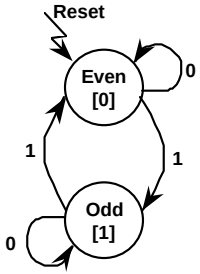
Computer Hardware = Datapath + Control



Concept of the State Machine

Example: Odd Parity Checker

Assert output whenever input bit stream has odd # of 1's



State Diagram

Present State	Input	Next State	Output
Even	0	Even	0
Even	1	Odd	0
Odd	0	Odd	1
Odd	1	Even	1

Symbolic State Transition Table

Present State	Input	Next State	Output
0	0	0	0
0	1	1	0
1	0	1	1
1	1	0	1

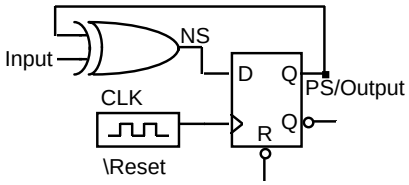
Encoded State Transition Table

Concept of the State Machine

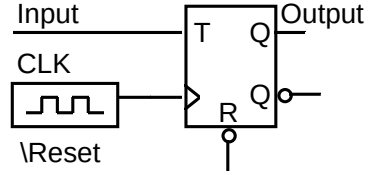
Example: Odd Parity Checker

Next State/Output Functions

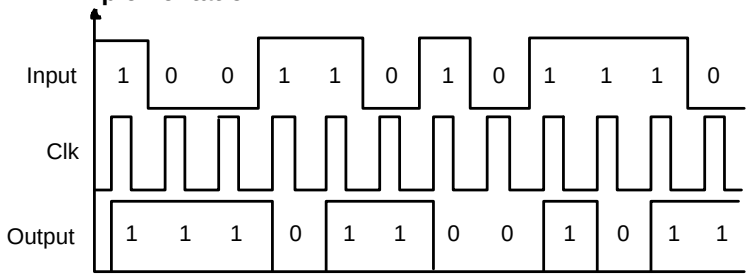
NS = PS xor PI; OUT = PS



D FF Implementation



T FF Implementation



Timing Behavior: Input 1 0 0 1 1 0 1 0 1 1 1 0

Concept of State Machine

Timing:
When are inputs sampled, next state computed, outputs asserted?

State Time: **Time between clocking events**

" Clocking event causes state/outputs to transition, based on inputs

" For set-up/hold time considerations:

Inputs should be stable before clocking event

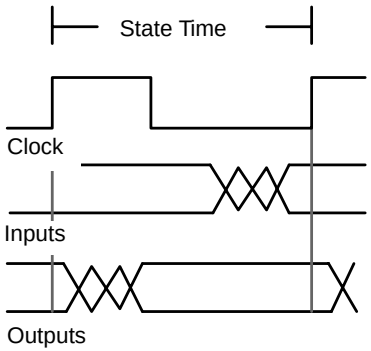
" After propagation delay, Next State entered, Outputs are stable

NOTE: Asynchronous signals take effect immediately
Synchronous signals take effect at the next clocking event

E.g., tri-state enable: effective immediately
sync. counter clear: effective at next clock event

Concept of State Machine

Example: **Positive Edge Triggered Synchronous System**



On rising edge, inputs sampled
outputs, next state computed

After propagation delay, outputs and
next state are stable

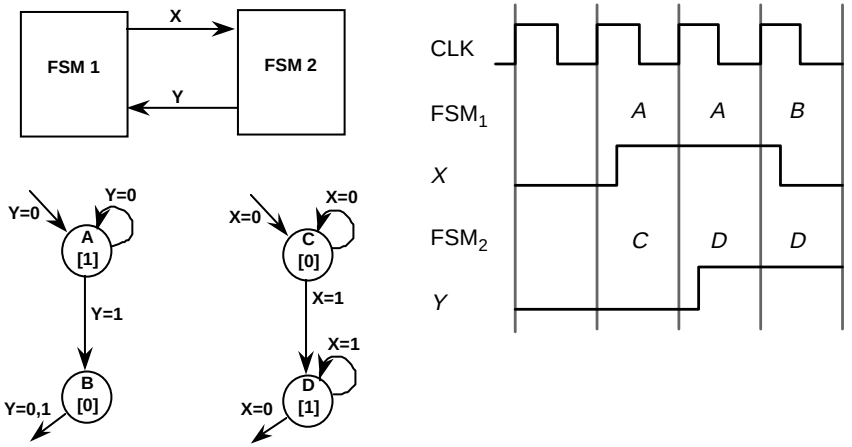
Immediate Outputs:
affect datapath immediately
could cause inputs from datapath to change

Delayed Outputs:
take effect on next clock edge
propagation delays must exceed hold times

Concept of the State Machine

Communicating State Machines

One machine's output is another machine's input



Machines advance in lock step

Initial inputs/outputs: X = 0, Y = 0

Basic Design Approach

Six Step Process

1. **Understand the statement of the Specification**
2. **Obtain an abstract specification of the FSM**
3. **Perform a state mininimization**
4. **Perform state assignment**
5. **Choose FF types to implement FSM state register**
6. **Implement the FSM**

1, 2 covered now; 3, 4, 5 covered later;
4, 5 generalized from the counter design procedure

Basic Design Approach

Example: **Vending Machine FSM**

General Machine Concept:

deliver package of gum after 15 cents deposited

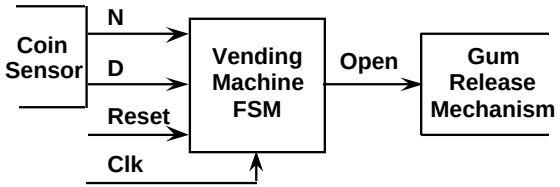
single coin slot for dimes, nickels

no change

Step 1. ***Understand the problem:***

Draw a picture!

Block Diagram



Vending Machine Example

Step 2. ***Map into more suitable abstract representation***

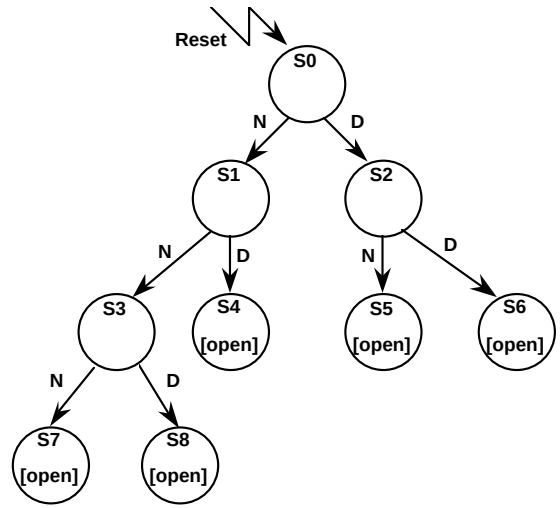
Tabulate typical input sequences:

three nickels
nickel, dime
dime, nickel
two dimes
two nickels, dime

Draw state diagram:

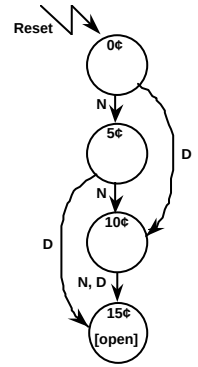
Inputs: N, D, reset

Output: open



Vending Machine Example

Step 3: State Minimization



reuse states whenever possible

Present State	Inputs D	N	Next State	Output Open
0¢	0	0	0¢	0
	0	1	5¢	0
	1	0	10¢	0
	1	1	X	X
5¢	0	0	5¢	0
	0	1	10¢	0
	1	0	15¢	0
	1	1	X	X
10¢	0	0	10¢	0
	0	1	15¢	0
	1	0	15¢	0
	1	1	X	X
15¢	X	X	15¢	1

Symbolic State Table

Vending Machine Example

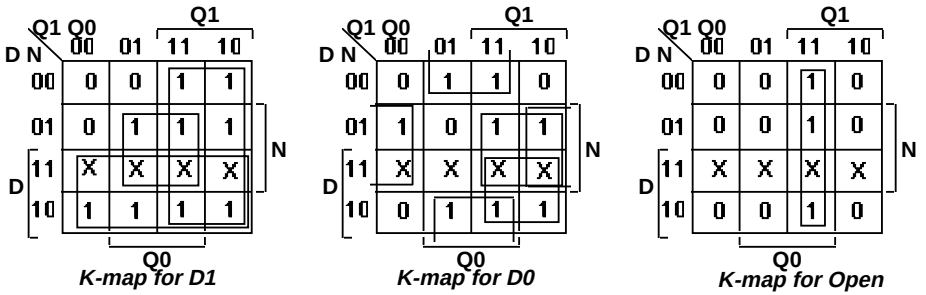
Step 4: State Encoding

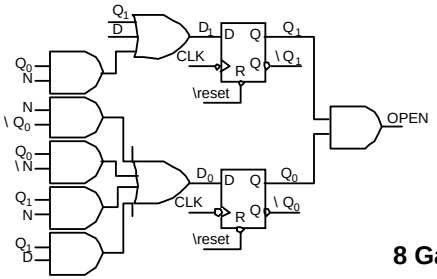
Present State Q_1	Q_0	Inputs D	N	Next State D_1	D_0	Output Open
0	0	0	0	0	0	0
		0	1	0	1	0
		1	0	1	0	0
		1	1	X	X	X
0	1	0	0	0	1	0
		0	1	1	0	0
		1	0	1	1	0
		1	1	X	X	X
1	0	0	0	1	0	0
		0	1	1	1	0
		1	0	1	1	0
		1	1	X	X	X
1	1	0	0	1	1	1
		0	1	1	1	1
		1	0	1	1	1
		1	1	X	X	X

Parity Checker Example

Step 5. *Choose FFs for implementation*

D FF easiest to use





D1 = Q1 + D + Q0 N

D0 = N $\overline{Q0}$ + Q0 $\overline{N}$ + Q1 N + Q1 D

OPEN = Q1 Q0

8 Gates

Parity Checker Example

Step 5. *Choosing FF for Implementation*

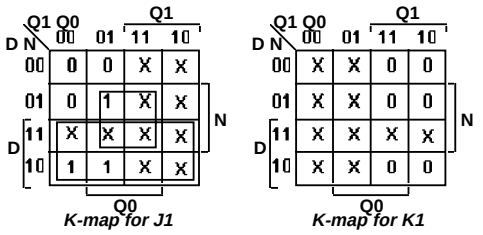
J-K FF

Present State Q_1	Q_0	Inputs D	N	Next State D_1	D_0	J_1	K_1	J_0	K_0
0	0	0	0	0	0	0	X	0	X
		0	1	0	1	0	X	1	X
		1	0	1	0	1	X	0	X
		1	1	X	X	X	X	X	X
0	1	0	0	0	1	0	X	X	0
		0	1	1	0	1	X	X	1
		1	0	1	1	1	X	X	0
		1	1	X	X	X	X	X	X
1	0	0	0	1	0	X	0	0	X
		0	1	1	1	X	0	1	X
		1	0	1	1	X	0	1	X
		1	1	X	X	X	X	X	X
1	1	0	0	1	1	X	0	X	0
		0	1	1	1	X	0	X	0
		1	0	1	1	X	0	X	0
		1	1	X	X	X	X	X	X

Remapped encoded state transition table

Vending Machine Example

Implementation:

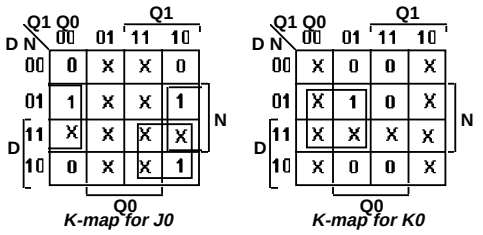


K-map for J1

K-map for K1

J1 = D + Q0 N

K1 = 0

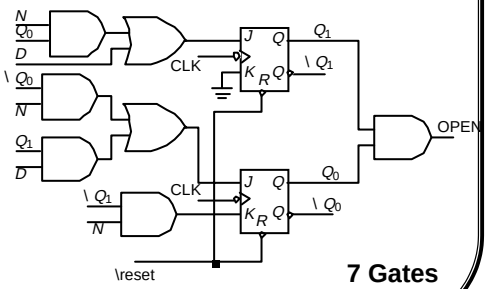


K-map for J0

K-map for K0

J0 = $\overline{Q0}$ N + Q1 D

K0 = $\overline{Q1}$ N



7 Gates

Alternative State Machine Representations

Why State Diagrams Are Not Enough

Not flexible enough for describing very complex finite state machines

Not suitable for gradual refinement of finite state machine

Do not obviously describe an *algorithm:* that is, well specified sequence of actions based on input data

algorithm = sequencing + data manipulation

separation of control and data

Gradual shift towards program-like representations:

" Algorithmic State Machine (ASM) Notation

" Hardware Description Languages (e.g., VHDL)

Alternative State Machine Representations

Algorithmic State Machine (ASM) Notation

Three Primitive Elements:

" State Box

" Decision Box

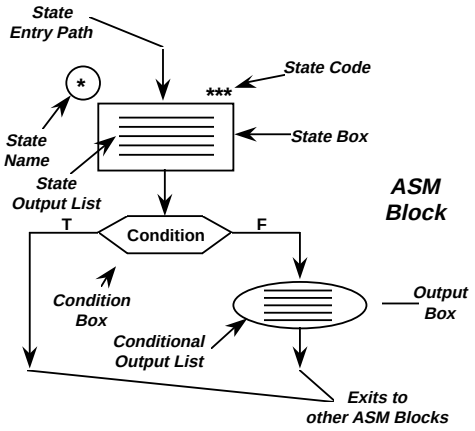
" Output Box

State Machine in one state block per state time

Single Entry Point

Unambiguous Exit Path for each combination of inputs

Outputs asserted high (.H) or low (.L); Immediate (I) or delayed til next clock



Alternative State Machine Representations

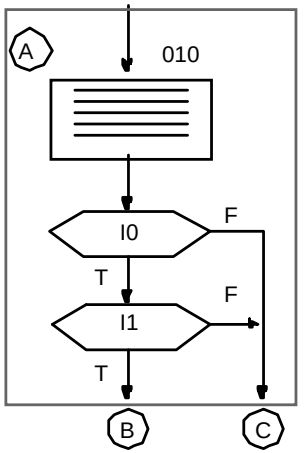
ASM Notation

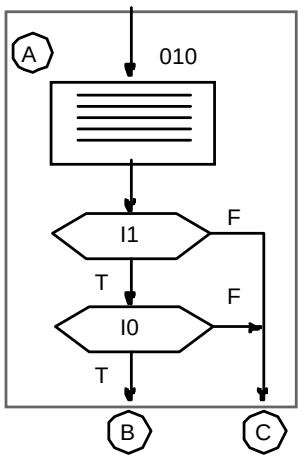
Condition Boxes:

Ordering has no effect on final outcome

Equivalent ASM charts:

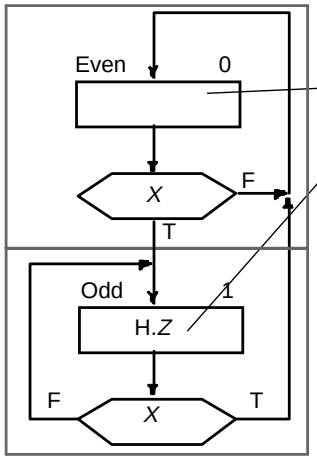
A exits to B on (I0 " I1) else exit to C





Alternative State Machine Representations

Example: Parity Checker



Input X, Output Z

Nothing in output list implies Z not asserted

Z asserted in State Odd

Symbolic State Table:

Input	*Present State*	*Next State*	*Output*
F	Even	Even	—
T	Even	Odd	—
F	Odd	Odd	A
T	Odd	Even	A

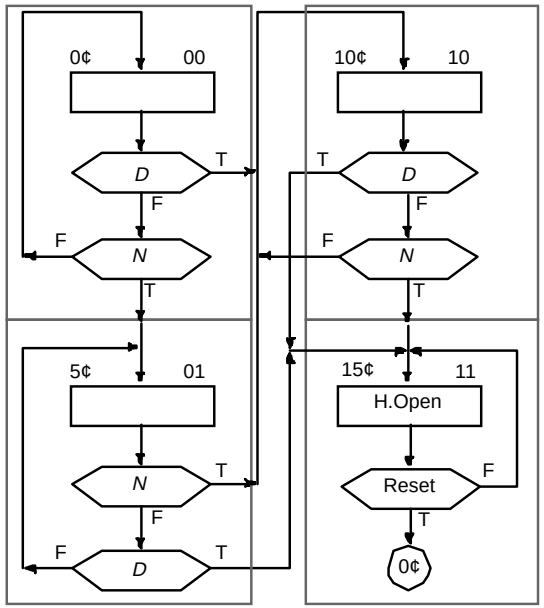
Encoded State Table:

Input	*Present State*	*Next State*	*Output*
0	0	0	0
1	0	1	0
0	1	1	1
1	1	0	1

Trace paths to derive state transition tables

Alternative State Machine Representations

ASM Chart for Vending Machine



Alternative State Machine Representations

Hardware Description Languages: VHDL

```
ENTITY parity_checker IS
 PORT (
  x, clk: IN BIT;
  z: OUT BIT);
END parity_checker;

ARCHITECTURE behavioral OF parity_checker IS
 BEGIN
  main: BLOCK (clk = '1' and not clk'STABLE)

  TYPE state IS (Even, Odd);
  SIGNAL state_register: state := Even;

  BEGIN state_even:
   BLOCK ((state_register = Even) AND GUARD)
    BEGIN
     state_register <= Odd WHEN x = '1'
      ELSE Even
   END BLOCK state_even;

  BEGIN state_odd:
   BLOCK ((state_register = Odd) AND GUARD)
    BEGIN
     state_register <= Even WHEN x = '1'
      ELSE Odd;
   END BLOCK state_odd;

  z <= '0' WHEN state_register = Even ELSE
     '1' WHEN state_register = Odd;
 END BLOCK main;
END behavioral;
```

Interface Description

Architectural Body

Guard Expression

Determine New State

Determine Outputs

Alternative State Machine Representations

ABEL Hardware Description Language

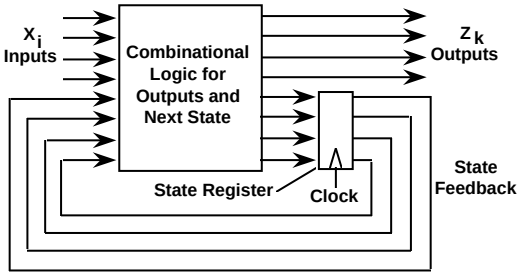
```
module parity
title 'odd parity checker state machine'
u1 device 'p22v10';

"Input Pins
 clk, X, RESET pin 1, 2, 3;

"Output Pins
 Q, Z    pin 21, 22;

 Q, Z    istype 'pos,reg';

"State registers
SREG  =  [Q, Z];
S0  =  [0, 0]; " even number of 0's
S1   =  [1, 1];  " odd number of 0's

equations
 [Q.ar, Z.ar] = RESET; "Reset to state S0
```

```
state_diagram SREG
state S0:
 if X then S1
 else S0;
state S1:
 if X then S0
 else S1;
```

```
test_vectors ([clk, RESET, X] -> [SREG])
 [0,1,.X.] -> [S0];
 [.C.,0,1] -> [S1];
 [.C.,0,1] -> [S0];
 [.C.,0,1] -> [S1];
 [.C.,0,0] -> [S1];
 [.C.,0,1] -> [S0];
 [.C.,0,1] -> [S1];
 [.C.,0,0] -> [S1];
 [.C.,0,0] -> [S1];
 [.C.,0,0] -> [S1];
end parity;
```

Moore and Mealy Machine Design Procedure

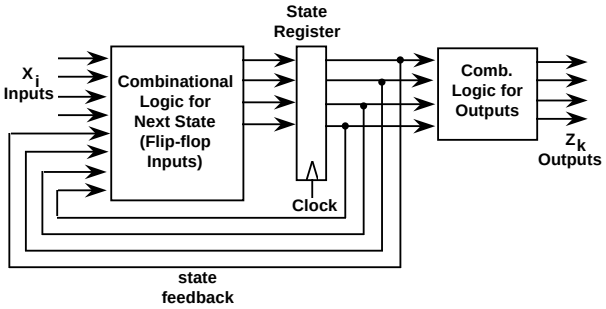
Definitions



Moore Machine

Outputs are function solely of the current state

Outputs change synchronously with state changes



Mealy Machine

Outputs depend on state AND inputs

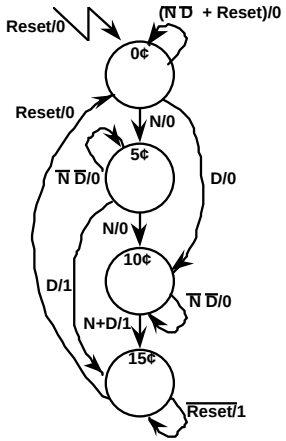
Input change causes an immediate output change

Asynchronous signals

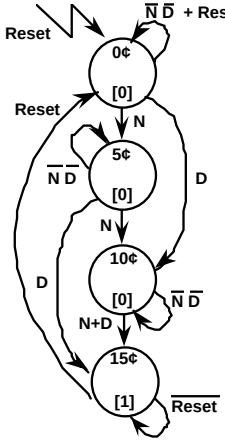
Moore and Mealy Machines

State Diagram Equivalents

Moore Machine



Outputs are associated with State



Mealy Machine

Outputs are associated with Transitions

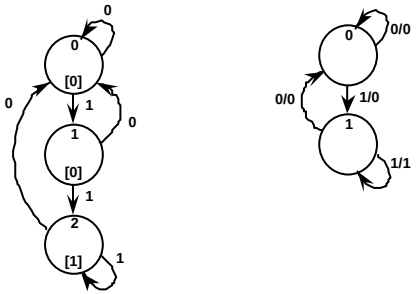
Moore and Mealy Machines

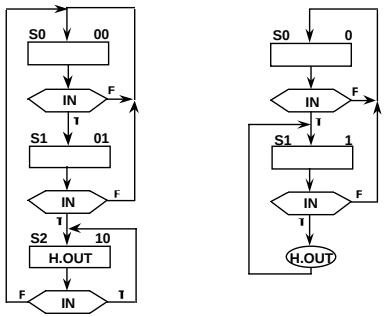
States vs. Transitions

Mealy Machine typically has fewer states than Moore Machine for same output sequence

Same I/O behavior

Different # of states



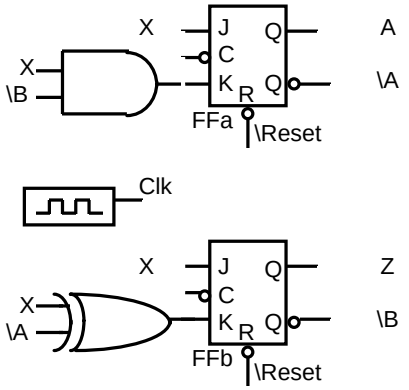


Equivalent ASM Charts

Moore and Mealy Machines

Timing Behavior of Moore Machines

Reverse engineer the following:



Input X
Output Z
State A, B = Z

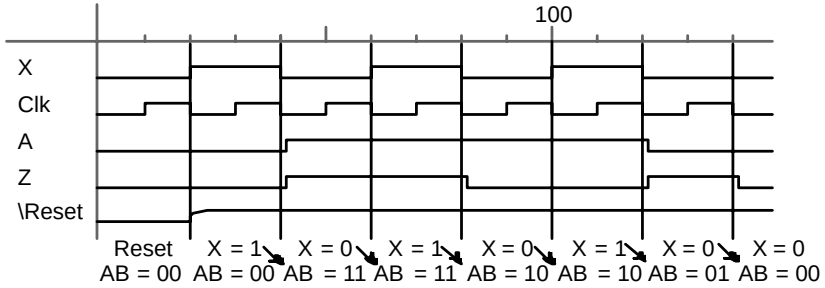
Two Techniques for Reverse Engineering:

- **Ad Hoc: Try input combinations to derive transition table**
- **Formal: Derive transition by analyzing the circuit**

Moore and Mealy Machines

Ad Hoc Reverse Engineering

Behavior in response to input sequence 1 0 1 0 1 0:



Partially Derived State Transition Table

A	B	X	A+	B+	Z
0	0	0	?	?	0
		1	1	1	0
0	1	0	0	0	1
		1	?	?	1
1	0	0	1	0	0
		1	0	1	0
1	1	0	1	1	1
		1	1	0	1

Moore and Mealy Machines

Formal Reverse Engineering

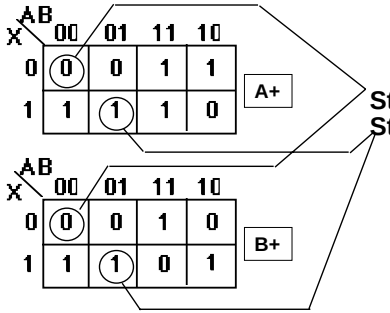
Derive transition table from next state and output combinational functions presented to the flipflops!

Ja = X **Ka = X " $\overline{B}$** **Z = B**
Jb = X **Kb = X xor A**

FF excitation equations for J-K flipflop:

A+ = Ja " $\overline{A}$ + $\overline{Ka}$ " A = X " $\overline{A}$ + ($\overline{X}$ + B) " A
B+ = Jb " B + Kb " B = X " B + (X " A + X " A) " B

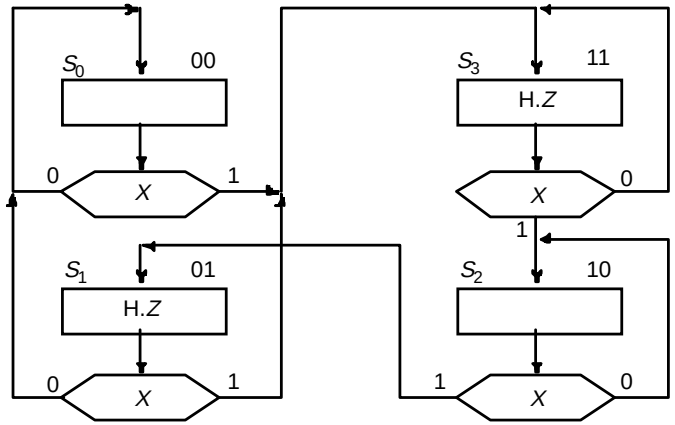
Next State K-Maps:



State 00, Input 0 -> State 00
State 01, Input 1 -> State 01

Moore and Mealy Machines

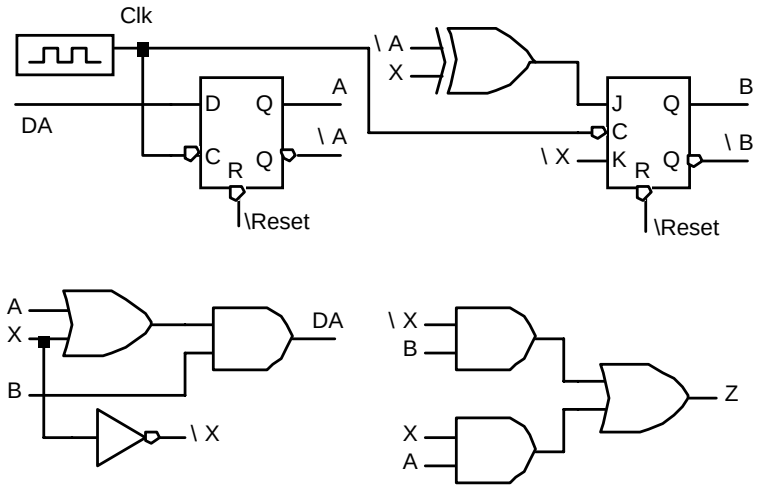
Complete ASM Chart for the Mystery Moore Machine



Note: All Outputs Associated With State Boxes
No Separate Output Boxes — Intrinsic in Moore Machines

Moore and Mealy Machines

Reverse Engineering a Mealy Machine



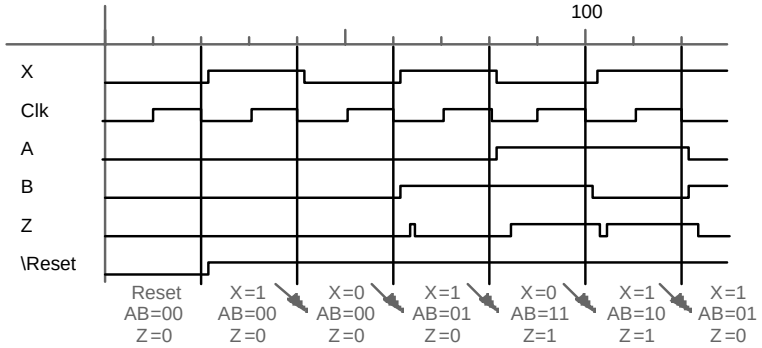
Input X, Output Z, State A, B

State register consists of D FF and J-K FF

Moore and Mealy Machine

Ad Hoc Method

Signal Trace of Input Sequence 101011:



Note glitches in Z!

Outputs valid at following falling clock edge

Partially completed state transition table based on the signal trace

A	B	X	A+	B+	Z
0	0	0	0	1	0
		1	0	0	0
0	1	0	?	?	?
		1	1	1	0
1	0	0	?	?	?
		1	0	1	1
1	1	0	1	0	1
		1	?	?	?

Moore and Mealy Machines

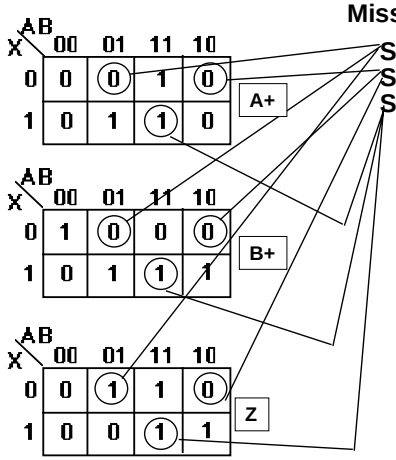
Formal Method

$A+ = B \cdot (A + X) = A \cdot B + B \cdot X$

$B+ = Jb \cdot \overline{B} + \overline{Kb} \cdot B = (\overline{A} \text{ xor } X) \cdot \overline{B} + X \cdot B$

$= A \cdot \overline{B} \cdot X + \overline{A} \cdot \overline{B} \cdot \overline{X} + B \cdot X$

$Z = A \cdot X + B \cdot \overline{X}$



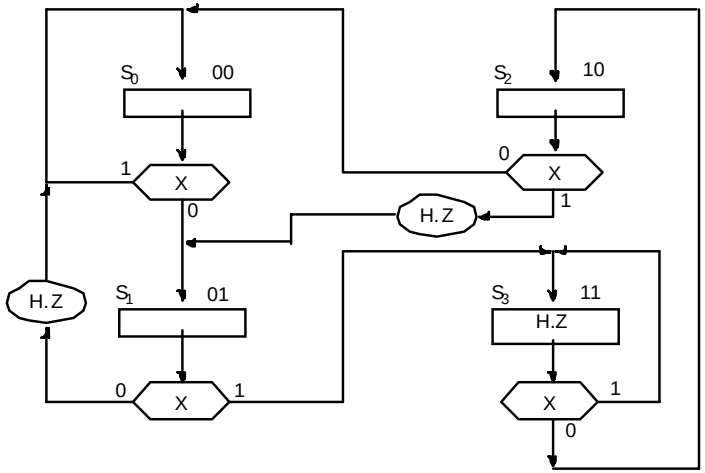
Missing Transitions and Outputs:

State 01, Input 0 -> State 01, Output 1
State 10, Input 0 -> State 00, Output 0
State 11, Input 1 -> State 11, Output 1

Moore and Mealy Machines

ASM Chart for Mystery Mealy Machine

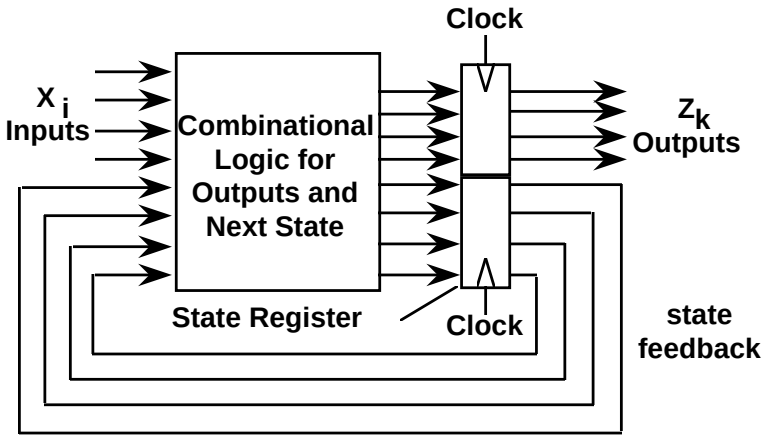
S0 = 00, S1 = 01, S2 = 10, S3 = 11



NOTE: Some Outputs in Output Boxes as well as State Boxes
This is intrinsic in Mealy Machine implementation

Moore and Mealy Machines

Synchronous Mealy Machine



latched state AND outputs

avoids glitchy outputs!

Finite State Machine Word Problems

Mapping English Language Description to Formal Specifications

Four Case Studies:

- **Finite String Pattern Recognizer**
- **Complex Counter with Decision Making**
- **Traffic Light Controller**
- **Digital Combination Lock**

We will use state diagrams and ASM Charts

Finite State Machine Word Problems

Finite String Pattern Recognizer

A finite string recognizer has one input (X) and one output (Z). The output is asserted whenever the input sequence &010& has been observed, as long as the sequence 100 has never been seen.

Step 1. Understanding the problem statement

Sample input/output behavior:

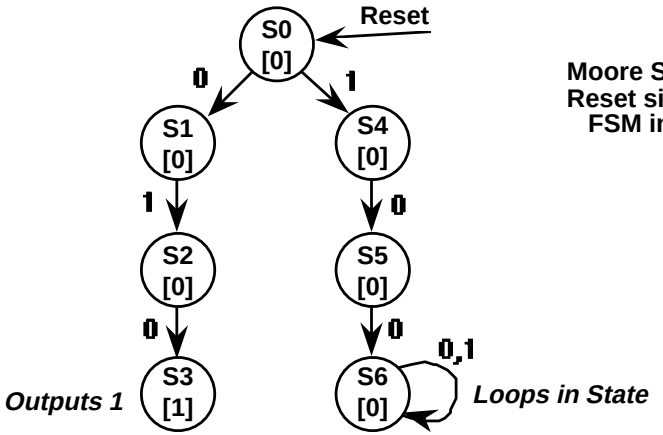
X: 00101010010&
Z: 00010101000&

X: 11011010010&
Z: 00000001000&

Finite State Machine Word Problems

Finite String Recognizer

Step 2. Draw State Diagrams/ASM Charts for the strings that must be recognized. I.e., 010 and 100.

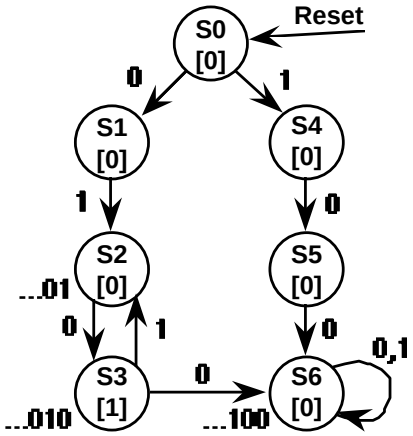


Moore State Diagram
Reset signal places FSM in S0

Finite State Machine Word Problems

Finite String Recognizer

Exit conditions from state S3: have recognized &010
if next input is 0 then have &0100!
if next input is 1 then have &0101 = &01 (state S2)

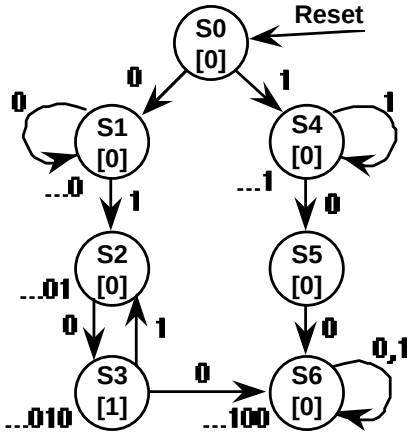


Finite State Machine Word Problems

Finite String Recognizer

Exit conditions from S1: recognizes strings of form &0 (no 1 seen)
loop back to S1 if input is 0

Exit conditions from S4: recognizes strings of form &1 (no 0 seen)
loop back to S4 if input is 1



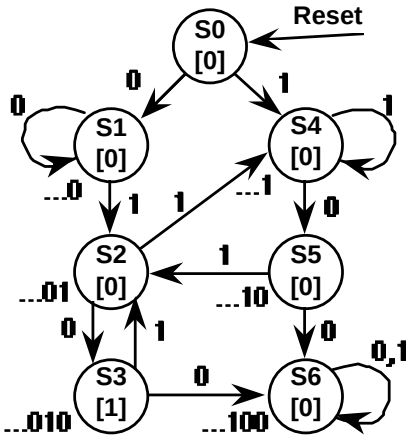
Finite State Machine Word Problems

Finite String Recognizer

S2, S5 with incomplete transitions

S2 = &01; If next input is 1, then string could be prefix of (01)1(00)
S4 handles just this case!

S5 = &10; If next input is 1, then string could be prefix of (10)1(0)
S2 handles just this case!



Final State Diagram

Finite State Machine Word Problems

Finite String Recognizer

```
module string
title '010/100 string recognizer state machine
  Josephine Engineer, Itty Bity Machines, Inc.'
u1 device 'p22v10';

"Input Pins
 clk, X, RESET    pin 1, 2, 3;

"Output Pins
 Q0, Q1, Q2, Z    pin 19, 20, 21, 22;

 Q0, Q1, Q2, Z istype 'pos,reg';

"State registers
SREG  =  [Q0, Q1, Q2, Z];
S0  =  [0,0,0,0]; " Reset state
S1  =  [0,0,1,0]; " strings of the form ...0
S2 = [0,1,0,0]; " strings of the form ...01
S3 = [0,1,1,1]; " strings of the form ...010
S4 = [1,0,0,0]; " strings of the form ...1
S5 = [1,0,1,0]; " strings of the form ...10
S6 = [1,1,0,0]; " strings of the form ...100

equations
 [Q0.ar, Q1.ar, Q2.ar, Z.ar] = RESET; "Reset to S0
```

```
state_diagram SREG
state S0: if X then S4 else S1;
state S1: if X then S2 else S1;
state S2: if X then S4 else S3;
state S3: if X then S2 else S6;
state S4: if X then S4 else S5;
state S5: if X then S2 else S6;
state S6: goto S6;

test_vectors ([clk, RESET, X] -> [Z]
 [0,1,.X.] -> [0];
 [.C.,0,0] -> [0];
 [.C.,0,0] -> [0];
 [.C.,0,1] -> [0];
 [.C.,0,0] -> [1];
 [.C.,0,1] -> [0];
 [.C.,0,0] -> [1];
 [.C.,0,1] -> [0];
 [.C.,0,0] -> [1];
 [.C.,0,0] -> [0];
 [.C.,0,1] -> [0];
 [.C.,0,0] -> [0];
end string;
```

ABEL Description

Finite State Machine Word Problems

Finite String Recognizer

Review of Process:

- **Write down sample inputs and outputs to understand specification**
- **Write down sequences of states and transitions for the sequences to be recognized**
- **Add missing transitions; reuse states as much as possible**
- **Verify I/O behavior of your state diagram to insure it functions like the specification**

Finite State Machine Word Problems

Complex Counter

A sync. 3 bit counter has a mode control M. When M = 0, the counter counts up in the binary sequence. When M = 1, the counter advances through the Gray code sequence.

Binary: 000, 001, 010, 011, 100, 101, 110, 111
Gray: 000, 001, 011, 010, 110, 111, 101, 100

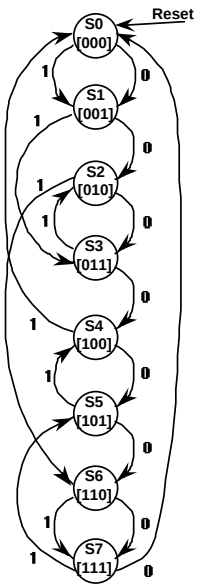
Valid I/O behavior:

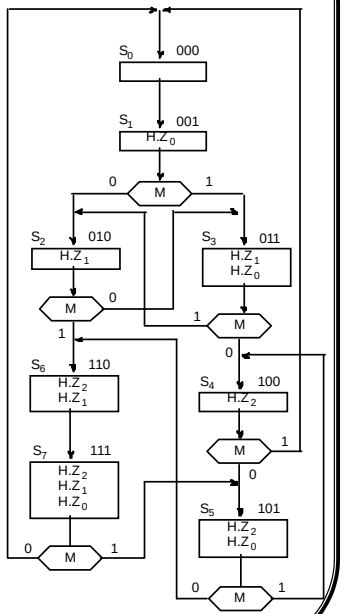
Mode Input M	Current State	Next State (Z2 Z1 Z0)
0	000	001
0	001	010
1	010	110
1	110	111
1	111	101
0	101	110
0	110	111

Finite State Machine Word Problems

Complex Counter

One state for each output combination
Add appropriate arcs for the mode control





Finite State Machine Word Problems

Complex Counter

```
module counter
title 'combination binary/gray code upcounter
  Josephine Engineer, Itty Bity Machines, Inc.'
u1 device 'p22v10';

"Input Pins                          state_diagram SREG
 clk, M, RESET    pin 1, 2, 3;       state S0: goto S1;
                                     state S1: if M then S3 else S2;
                                     state S2: if M then S6 else S3;
"Output Pins                         state S3: if M then S2 else S4;
 Z0, Z1, Z2    pin 19, 20, 21;       state S4: if M then S0 else S5;
                                     state S5: if M then S4 else S6;
 Z0, Z1, Z2   istype 'pos,reg';      state S6: goto S7;
                                     state S7: if M then S5 else S0;
"State registers
SREG  =  [Z0, Z1, Z2];               test_vectors ([clk, RESET, M] -> [Z0, Z1, Z2])
S0  =  [0,0,0];                       [0,1,.X.] -> [0,0,0];
S1  =  [0,0,1];                       [.C.,0,0] -> [0,0,1];
S2 = [0,1,0];                         [.C.,0,0] -> [0,1,0];
S3 = [0,1,1];                         [.C.,0,1] -> [1,1,0];
S4 = [1,0,0];                         [.C.,0,1] -> [1,1,1];
S5 = [1,0,1];                         [.C.,0,1] -> [1,0,1];
S6 = [1,1,0];                         [.C.,0,0] -> [1,1,0];
S7 = [1,1,1];                         [.C.,0,0] -> [1,1,1];
                                     end counter;
equations
 [Z0.ar, Z1.ar, Z2.ar] = RESET; "Reset to state S0
```

ABEL Description

Finite State Machine Word Problems

Traffic Light Controller

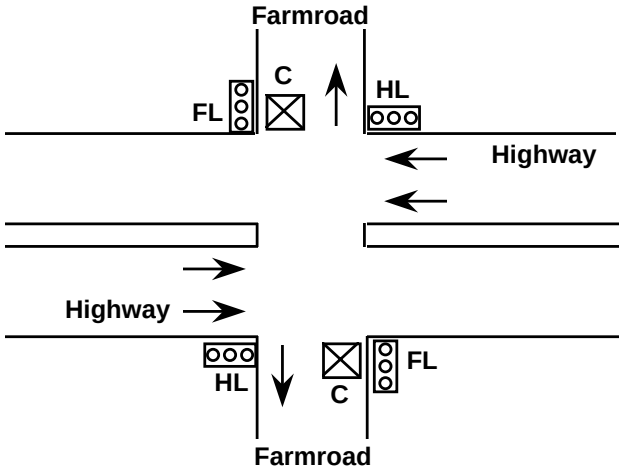
A busy highway is intersected by a little used farmroad. Detectors C sense the presence of cars waiting on the farmroad. With no car on farmroad, light remain green in highway direction. If vehicle on farmroad, highway lights go from Green to Yellow to Red, allowing the farmroad lights to become green. These stay green only as long as a farmroad car is detected but never longer than a set interval. When these are met, farm lights transition from Green to Yellow to Red, allowing highway to return to green. Even if farmroad vehicles are waiting, highway gets at least a set interval as green.

Assume you have an interval timer that generates a short time pulse (TS) and a long time pulse (TL) in response to a set (ST) signal. TS is to be used for timing yellow lights and TL for green lights.

Finite State Machine Word Problems

Traffic Light Controller

Picture of Highway/Farmroad Intersection:



Finite State Machine Word Problems

Traffic Light Controller

" **Tabulation of Inputs and Outputs:**

Input Signal	*Description*
reset	place FSM in initial state
C	detect vehicle on farmroad
TS	short time interval expired
TL	long time interval expired

Output Signal	*Description*
HG, HY, HR	assert green/yellow/red highway lights
FG, FY, FR	assert green/yellow/red farmroad lights
ST	start timing a short or long interval

" **Tabulation of Unique States: Some light configuration imply others**

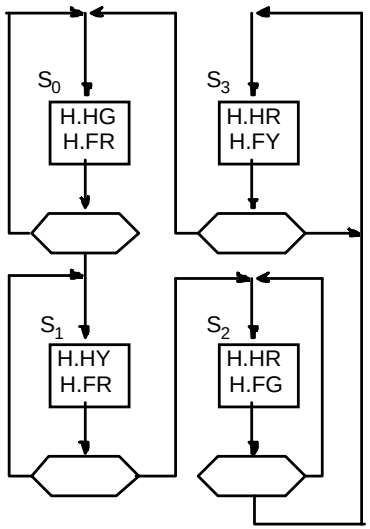
State	*Description*
S0	Highway green (farmroad red)
S1	Highway yellow (farmroad red)
S2	Farmroad green (highway red)
S3	Farmroad yellow (highway red)

Finite State Machine Word Problems

Traffic Light Controller

Refinement of ASM Chart:

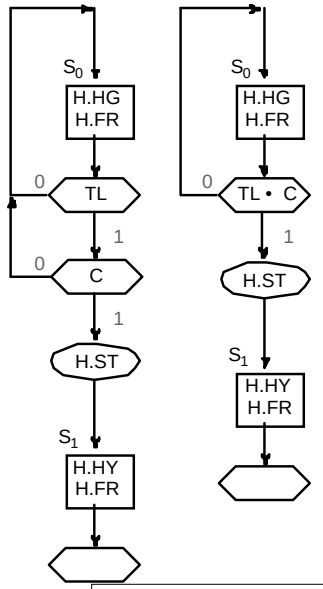
Start with basic sequencing and outputs:



Finite State Machine Word Problems

Traffic Light Controller

Determine Exit Conditions for S0:
Car waiting and Long Time Interval Expired- C " TL

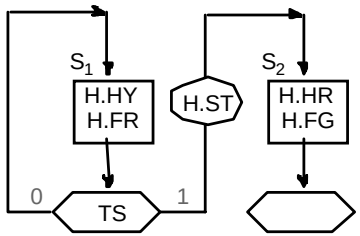


Equivalent ASM Chart Fragments

Finite State Machine Word Problems

Traffic Light Controller

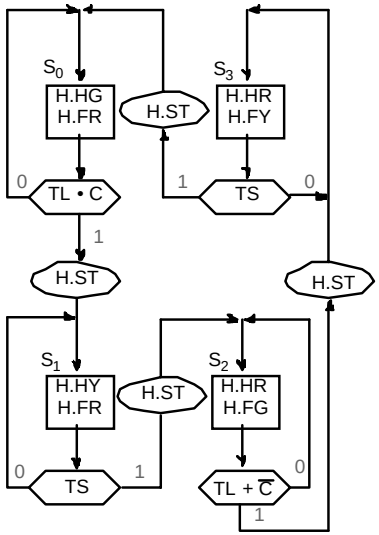
S1 to S2 Transition:
- **Set ST on exit from S0**
- **Stay in S1 until TS asserted**
- **Similar situation for S3 to S4 transition**



Finite State Machine Word Problems

Traffic Light Controller

S2 Exit Condition: no car waiting OR long time interval expired

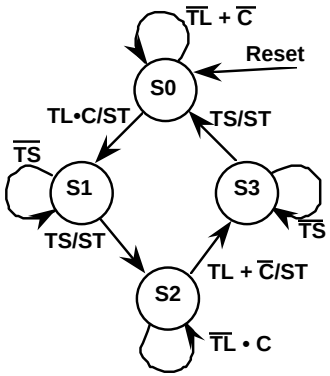


Complete ASM Chart for Traffic Light Controller

Finite State Machine Word Problems

Traffic Light Controller

Compare with state diagram:



S0: HG

S1: HY

S2: FG

S3: FY

Advantages of State Charts:

- **Concentrates on paths and conditions for exiting a state**
- **Exit conditions built up incrementally, later combined into single Boolean condition for exit**
- **Easier to understand the design as an algorithm**

Finite State Machine Word Problems

Traffic Light Controller

```
module traffic
title 'traffic light FSM'
u1 device 'p22v10';

"Input Pins
 clk, C, RESET, TS, TL
pin   1, 2,     3,  4,  5;

"Output Pins
 Q0, Q1, HG, HY, HR,
 FG, FY, FR, ST
pin 14, 15, 16, 17, 18,
    19, 20, 21, 22;

 Q0, Q1 istype 'pos,reg';
 ST, HG, HY, HR,
 FG, FY, FR istype 'pos,com';

"State registers
SREG  =  [Q0, Q1];
S0  =  [ 0,  0];
S1  =  [ 0,  1];
S2 = [ 1,  0];
S3 = [ 1,  1];

equations
 [Q0.ar, Q1.ar] = RESET;
 HG = !Q0 & !Q1;
 HY = !Q0 & Q1;
 HR = (Q0 & !Q1) # (Q0 & Q1);
 FG = Q0 & !Q1;
 FY = Q0 & Q1;
 FR = (!Q0 & !Q1) # (!Q0 & Q1);

state_diagram SREG
state S0:  if (TL & C) then S1 with ST = 1
   else S0 with ST = 0
state S1: if TS then S2 with ST = 1
   else S1 with ST = 0
state S2:  if (TL # !C) then S3 with ST = 1
   else S2 with ST = 0
state S3:  if TS then S0 with ST = 1
   else S3 with ST = 0

test_vectors
([clk,RESET, C, TS, TL]->[SREG,HG,HY,HR,FG,FY,FR,ST])
 [.X.,   1,.X.,.X.,.X.]->[ S0, 1, 0, 0, 0, 0, 1, 0];
 [.C.,   0,  0,  0,  0]->[ S0, 1, 0, 0, 0, 0, 1, 0];
 [.C.,   0,  1,  0,  1]->[ S1, 0, 1, 0, 0, 0, 1, 0];
 [.C.,   0,  1,  0,  0]->[ S1, 0, 1, 0, 0, 0, 1, 0];
 [.C.,   0,  1,  1,  0]->[ S2, 0, 0, 1, 1, 0, 0, 0];
 [.C.,   0,  1,  0,  0]->[ S2, 0, 0, 1, 1, 0, 0, 0];
 [.C.,   0,  1,  0,  1]->[ S3, 0, 0, 1, 0, 1, 0, 0];
 [.C.,   0,  1,  1,  0]->[ S0, 1, 0, 0, 0, 0, 1, 0];
end traffic;
```

ABEL Description

Finite State Machine Word Problems

Digital Combination Lock

"3 bit serial lock controls entry to locked room. Inputs are RESET, ENTER, 2 position switch for bit of key data. Locks generates an UNLOCK signal when key matches internal combination. ERROR light illuminated if key does not match combination. Sequence is: (1) Press RESET, (2) enter key bit, (3) Press ENTER, (4) repeat (2) & (3) two more times."

Problem specification is incomplete:

- **how do you set the internal combination?**
- **exactly when is the ERROR light asserted?**

Make reasonable assumptions:

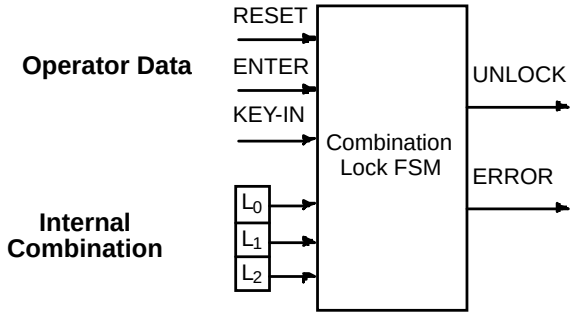
- **hardwired into next state logic vs. stored in internal register**
- **assert as soon as error is detected vs. wait until full combination has been entered**

Our design: registered combination plus error after full combination

Finite State Machine Word Problems

Digital Combination Lock

Understanding the problem: draw a block diagram &



Inputs:
Reset
Enter
Key-In
L0, L1, L2

Outputs:
Unlock
Error

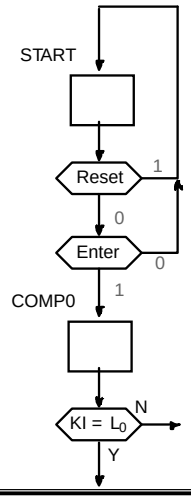
Finite State Machine Word Problems

Digital Combination Lock

Enumeration of states:

what sequences lead to opening the door?
error conditions on a second pass &

START state plus three key COMParison states



START entered on RESET

Exit START when ENTER is pressed

Continue on if Key-In matches L0

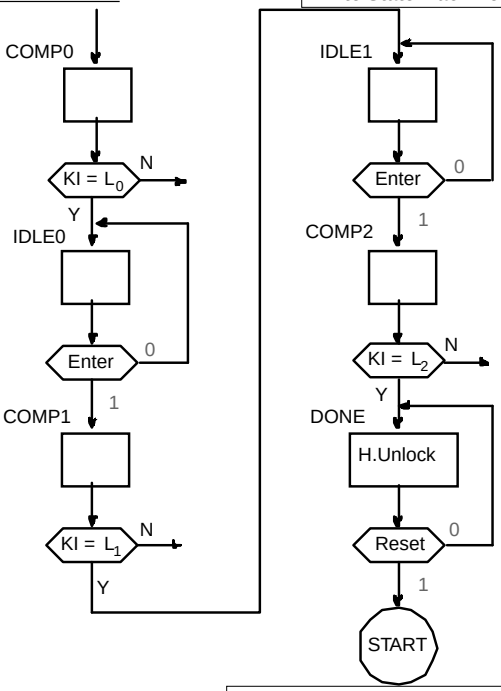
Finite State Machine Word Problems

Digital Combination Lock

Path to unlock:

Wait for Enter Key press

Compare Key-IN

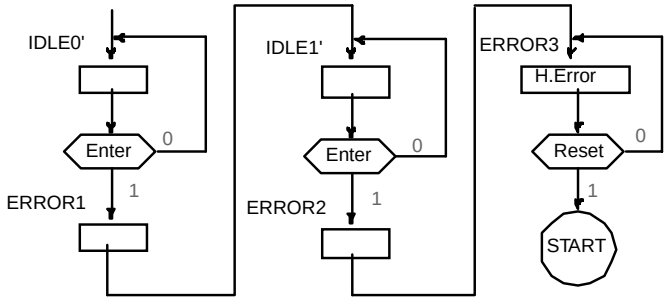


Finite State Machine Word Problems

Digital Combination Lock

Now consider error paths

Should follow a similar sequence as UNLOCK path, except asserting ERROR at the end:



COMP0 error exits to IDLE0'

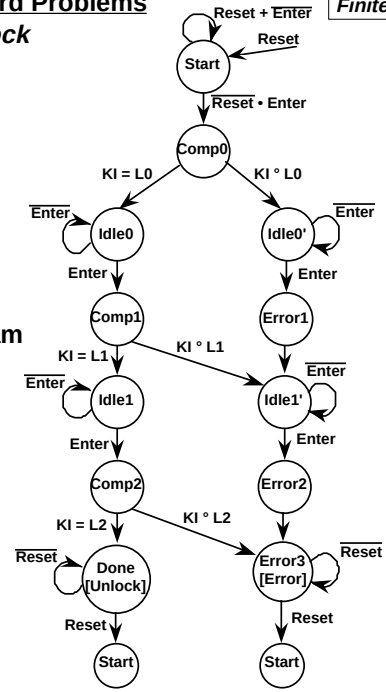
COMP1 error exits to IDLE1'

COMP2 error exits to ERROR3

Finite State Machine Word Problems

Digital Combination Lock

Equivalent State Diagram



Finite State Machine Word Problems

Combination Lock

```
module lock
title 'comb. lock FSM'
u1 device 'p22v10';

"Input Pins
clk, RESET, ENTER, L0, L1, L2, KI
pin   1, 2, 3, 4, 5, 6, 7;

"Output Pins
Q0, Q1, Q2, Q3, UNLOCK, ERROR
pin 16, 17, 18, 19, 14, 15;

Q0, Q1, Q2, Q3 istype 'pos,reg';
UNLOCK, ERROR  istype 'pos,com';

"State registers
SREG   =  [Q0, Q1, Q2, Q3];
START   =  [ 0,  0,  0,  0];
COMP0   =  [ 0,  0,  0,  1];
IDLE0  = [ 0,  0,  1,  0];
COMP1  = [ 0,  0,  1,  1];
IDLE1  = [ 0,  1,  0,  0];
COMP2  = [ 0,  1,  0,  1];
DONE  = [ 0,  1,  1,  0];
IDLE0p  = [ 0,  1,  1,  1];
ERROR1  = [ 1,  0,  0,  0];
IDLE1p  = [ 1,  0,  0,  1];
ERROR2  = [ 1,  0,  1,  0];
ERROR3  = [ 1,  0,  1,  1];

equations
 [Q0.ar, Q1.ar, Q2.ar, Q3.ar] = RESET;
 UNLOCK = !Q0 & Q1 & Q2 & !Q3;"asserted in DONE
 ERROR = Q0 & !Q1 & Q2 & Q3;  "asserted in ERROR3

state_diagram SREG
state START: if (RESET # !ENTER)
                then START else COMP0;
state COMP0: if (KI == L0) then IDLE0 else IDLE0p;
state IDLE0: if (!ENTER) then IDLE0 else COMP1;
state COMP1: if (KI == L1) then IDLE1 else IDLE1p;
state IDLE1: if (!ENTER) then IDLE1 else COMP2;
state COMP2: if (KI == L2) then DONE else ERROR3;
state DONE:  if (!RESET) then DONE else START;
state IDLE0p:if (!ENTER) then IDLE0p else ERROR1;
state ERROR1:goto IDLE1p;
state IDLE1p:if (!ENTER) then IDLE1p else ERROR2;
state ERROR2:goto ERROR3;
state ERROR3:if (!RESET) then ERROR3 else START;

test_vectors

end lock;
```


Chapter Review

Basic Timing Behavior an FSM

- **when are inputs sampled, next state/outputs transition and stabilize**
- **Moore and Mealy (Async and Sync) machine organizations**
  outputs = F(state) vs. outputs = F(state, inputs)

First Two Steps of the Six Step Procedure for FSM Design

- **understanding the problem**
- **abstract representation of the FSM**

Abstract Representations of an FSM

- **ASM Charts, Hardware Description Languages**

Word Problems

- **understand I/O behavior; draw diagrams**
- **enumerate states for the "goal"; expand with error conditions**
- **reuse states whenever possible**

Chapter #9: Finite State Machine Optimization

Contemporary Logic Design

Randy H. Katz
University of California, Berkeley

July 1993

Chapter Outline

" ***Procedures for optimizing implementation of an FSM***

State Reduction

State Assignment

" ***Computer Tools for State Assignment: Nova, Mustang, Jedi***

" ***Choice of Flipflops***

" ***FSM Partitioning***

Motivation

Basic FSM Design Procedure:

(1) Understand the problem

(2) Obtain a formal description

(3) Minimize number of states

(4) Encode the states

(5) Choose FFs to implement state register

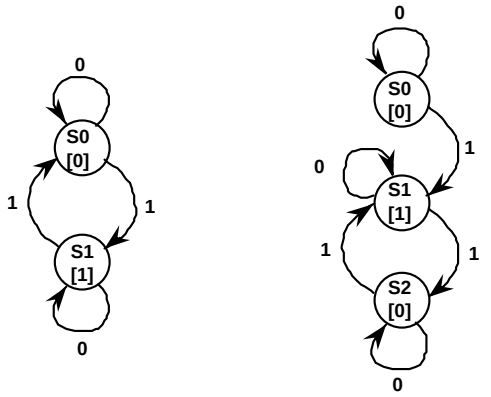
This Chapter!

(6) Implement the FSM

Next Chapter

Motivation

State Reduction



Odd Parity Checker: two alternative state diagrams

" **Identical output behavior on all input strings**

" **FSMs are *equivalent*, but require different implementations**

" **Design state diagram without concern for # of states, Reduce later**

Motivation

State Reduction (continued)

Implement FSM with fewest possible states

- Least number of flipflops
- Boundaries are power of two number of states
- Fewest states usually leads to more opportunities for don't cares
- Reduce the number of gates needed for implementation

State Reduction

Goal

Identify and combine states that have equivalent behavior

Equivalent States: for all input combinations, states transition to the same or equivalent states

Default Designer: S0, S2 are equivalent states
Both output a 0
Both transition to S1 on a 1 and self-loop on a 0

Algorithmic Approach

- Start with state transition table
- Identify states with same output behavior
- If such states transition to the same next state, they are equivalent
- Combine into a single new renamed state
- Repeat until no new states are combined

State Reduction

Row Matching Method

Example FSM Specification:

Single input X, output Z
Taking inputs grouped four at a time, output 1 if last four inputs were the string 1010 or 0110

Example I/O Behavior:

X = 0010 0110 1100 1010 0011 . . .
Z = 0000 0001 0000 0001 0000 . . .

Upper bound on FSM complexity:

Fifteen states (1 + 2 + 4 + 8)

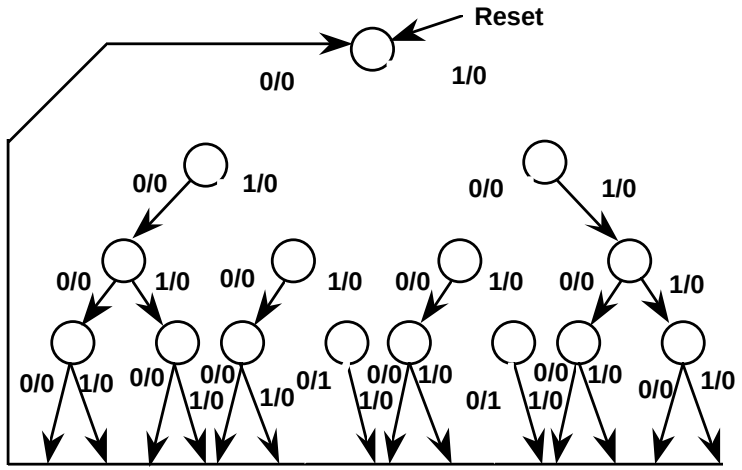
Thirty transitions (2 + 4 + 8 + 16)

sufficient to recognize any binary string of length four!

State Reduction

Row Matching Method

State Diagram for Example FSM:



State Reduction

Row Matching Method

Initial State Transition Table:

Input Sequence	Present State	Next State X=0	Next State X=1	Output X=0	Output X=1
Reset	S_0	S_1	S_2	0	0
0	S_1	S_3	S_4	0	0
1	S_2	S_5	S_6	0	0
00	S_3	S_7	S_8	0	0
01	S_4	S_9	S_{10}	0	0
10	S_5	S_{11}	S_{12}	0	0
11	S_6	S_{13}	S_{14}	0	0
000	S_7	S_0	S_0	0	0
001	S_8	S_0	S_0	0	0
010	S_9	S_0	S_0	0	0
011	S_{10}	S_0	S_0	1	0
100	S_{11}	S_0	S_0	0	0
101	S_{12}	S_0	S_0	1	0
110	S_{13}	S_0	S_0	0	0
111	S_{14}	S_0	S_0	0	0

State Reduction

Row Matching Method

Initial State Transition Table:

Input Sequence	Present State	Next State X=0	Next State X=1	Output X=0	Output X=1
Reset	S_0	S_1	S_2	0	0
0	S_1	S_3	S_4	0	0
1	S_2	S_5	S_6	0	0
00	S_3	S_7	S_8	0	0
01	S_4	S_9	S_{10}	0	0
10	S_5	S_{11}	S_{12}	0	0
11	S_6	S_{13}	S_{14}	0	0
000	S_7	S_0	S_0	0	0
001	S_8	S_0	S_0	0	0
010	S_9	S_0	S_0	0	0
011	S_{10}	S_0	S_0	1	0
100	S_{11}	S_0	S_0	0	0
101	S_{12}	S_0	S_0	1	0
110	S_{13}	S_0	S_0	0	0
111	S_{14}	S_0	S_0	0	0

State Reduction

Row Matching Method

Input Sequence	Present State	Next State X=0	Next State X=1	Output X=0	Output X=1
Reset	S_0	S_1	S_2	0	0
0	S_1	S_3	S_4	0	0
1	S_2	S_5	S_6	0	0
00	S_3	S_7	S_8	0	0
01	S_4	S_9	S'_{10}	0	0
10	S_5	S_{11}	S'_{10}	0	0
11	S_6	S_{13}	S_{14}	0	0
000	S_7	S_0	S_0	0	0
001	S_8	S_0	S_0	0	0
010	S_9	S_0	S_0	0	0
011 or 101	S'_{10}	S_0	S_0	1	0
100	S_{11}	S_0	S_0	0	0
110	S_{13}	S_0	S_0	0	0
111	S_{14}	S_0	S_0	0	0

State Reduction

Row Matching Method

Input Sequence	Present State	Next State X=0	Next State X=1	Output X=0	Output X=1
Reset	S_0	S_1	S_2	0	0
0	S_1	S_3	S_4	0	0
1	S_2	S_5	S_6	0	0
00	S_3	S_7	S_8	0	0
01	S_4	S_9	S'_{10}	0	0
10	S_5	S_{11}	S'_{10}	0	0
11	S_6	S_{13}	S_{14}	0	0
000	S_7	S_0	S_0	0	0
001	S_8	S_0	S_0	0	0
010	S_9	S_0	S_0	0	0
011 or 101	S'_{10}	S_0	S_0	1	0
100	S_{11}	S_0	S_0	0	0
110	S_{13}	S_0	S_0	0	0
111	S_{14}	S_0	S_0	0	0

State Reduction

Row Matching Method

Input Sequence	Present State	Next State X=0	Next State X=1	Output X=0	Output X=1
Reset	S_0	S_1	S_2	0	0
0	S_1	S_3	S_4	0	0
1	S_2	S_5	S_6	0	0
00	S_3	S'_7	S'_7	0	0
01	S_4	S'_7	S'_{10}	0	0
10	S_5	S'_7	S'_{10}	0	0
11	S_6	S'_7	S'_7	0	0
not (011 or 101)	S'_7	S_0	S_0	0	0
011 or 101	S'_{10}	S_0	S_0	1	0

State Reduction

Row Matching Method

Input Sequence	Present State	Next State X=0	Next State X=1	Output X=0	Output X=1
Reset	S_0	S_1	S_2	0	0
0	S_1	S_3	S_4	0	0
1	S_2	S_5	S_6	0	0
00	S_3	S'_7	S'_7	0	0
01	S_4	S'_7	S'_{10}	0	0
10	S_5	S'_7	S'_{10}	0	0
11	S_6	S'_7	S'_7	0	0
not (011 or 101)	S'_7	S_0	S_0	0	0
011 or 101	S'_{10}	S_0	S_0	1	0

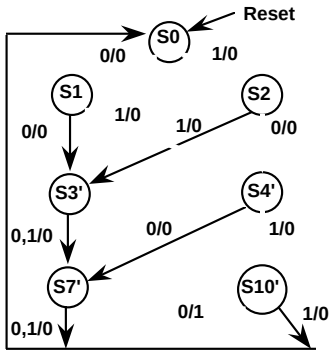
State Reduction

Row Matching Method

Final Reduced State Transition Table

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	S4'	S3'	0	0
00 or 11	S3'	S7'	S7'	0	0
01 or 10	S4'	S7'	S10'	0	0
not (011 or 101)	S7'	S0	S0	0	0
011 or 101	S10'	S0	S0	1	0

Corresponding State Diagram



State Reduction

Row Matching Method

- **Straightforward to understand and easy to implement**
- **Problem: does not allows yield the most reduced state table!**

Example: 3 State Odd Parity Checker

Present State	Next State X=0	Next State X=1	Output
S_0	S_0	S_1	0
S_1	S_1	S_2	1
S_2	S_2	S_1	0

No way to combine states S0 and S2 based on Next State Criterion!

State Reduction

Implication Chart Method

New example FSM:

Single input X, Single output Z

Output a 1 whenever the serial sequence 010 or 110 has been observed at the inputs

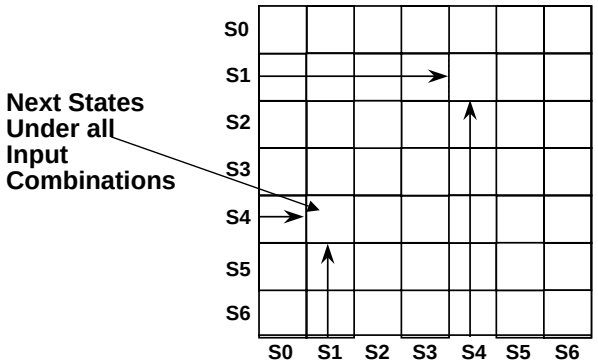
State transition table:

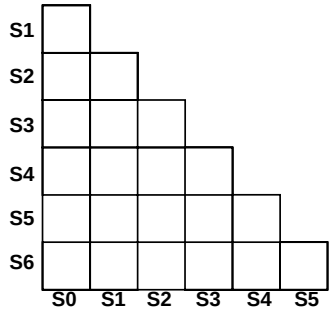
Input Sequence	Present State	Next State X=0	Next State X=1	Output X=0	Output X=1
Reset	S_0	S_1	S_2	0	0
0	S_1	S_3	S_4	0	0
1	S_2	S_5	S_6	0	0
00	S_3	S_0	S_0	0	0
01	S_4	S_0	S_0	1	0
10	S_5	S_0	S_0	0	0
11	S_6	S_0	S_0	1	0

State Reduction

Implication Chart Method

Enumerate all possible combinations of states taken two at a time





Implication Chart

Naive Data Structure:
Xij will be the same as Xji
Also, can eliminate the diagonal

State Reduction

Implication Chart

Filling in the Implication Chart

Entry Xij — Row is Si, Column is Sj

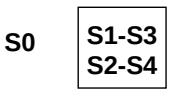
Si is equivalent to Sj if outputs are the same and next states are equivalent

Xij contains the next states of Si, Sj which must be equivalent if Si and Sj are equivalent

If Si, Sj have different output behavior, then Xij is crossed out

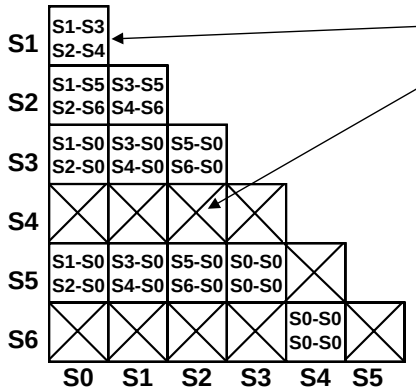
Example:
S0 transitions to S1 on 0, S2 on 1;
S1 transitions to S3 on 0, S4 on 1;

So square X<0,1> contains entries S1-S3 (transition on zero)
S2-S4 (transition on one)



State Reduction

Implication Chart Method



S2 and S4 have different I/O behavior

This implies that S1 and S0 cannot be combined

Starting Implication Chart

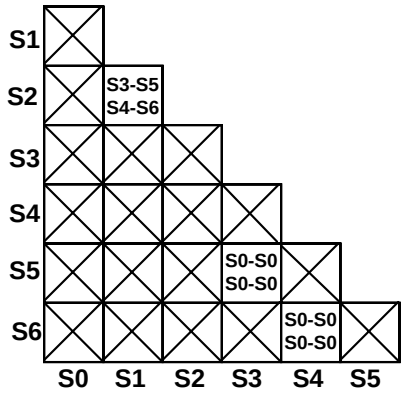
State Reduction

Implication Chart Method

Results of First Marking Pass

Second Pass Adds No New Information

S3 and S5 are equivalent
S4 and S6 are equivalent
This implies that S1 and S2 are too!

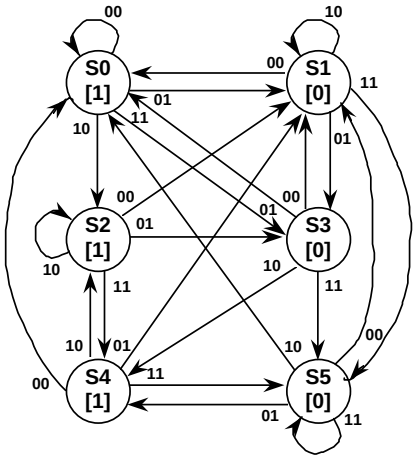


Reduced State Transition Table

Input Sequence	Present State	Next State X=0	Next State X=1	Output X=0	Output X=1
Reset	S_0	S_1'	S_1'	0	0
0 or 1	S_1'	S_3'	S_4'	0	0
00 or 10	S_3'	S_0	S_0	0	0
01 or 11	S_4'	S_0	S_0	1	0

State Reduction

Multiple Input State Diagram Example



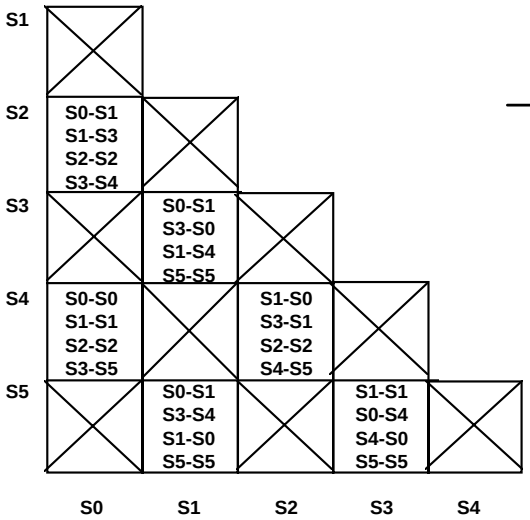
State Diagram

Present State	Next State 00	01	10	11	Output
S_0	S_0	S_1	S_2	S_3	1
S_1	S_0	S_3	S_1	S_5	0
S_2	S_1	S_3	S_2	S_4	1
S_3	S_1	S_0	S_4	S_5	0
S_4	S_0	S_1	S_2	S_5	1
S_5	S_1	S_4	S_0	S_5	0

Symbolic State Diagram

State Reduction

Multiple Input Example



Implication Chart

Present State	Next State 00	01	10	11	Output
S_0'	S_0'	S_1	S_2	S_3'	1
S_1	S_0'	S_3'	S_1	S_3'	0
S_2	S_1	S_3'	S_2	S_0'	1
S_3'	S_1	S_0'	S_0'	S_3'	0

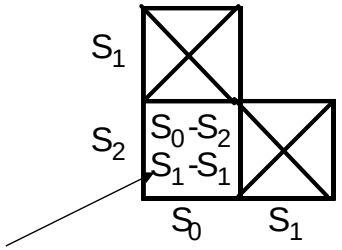
Minimized State Table

State Reduction

Implication Chart Method

Does the method solve the problem with the odd parity checker?

Implication Chart



S0 is equivalent to S2
since nothing contradicts this assertion!

State Reduction

Implication Chart Method

The detailed algorithm:

1. **Construct implication chart, one square for each combination of states taken two at a time**
2. **Square labeled Si, Sj, if outputs differ than square gets "X". Otherwise write down implied state pairs for all input combinations**
3. **Advance through chart top-to-bottom and left-to-right. If square Si, Sj contains next state pair Sm, Sn and that pair labels a square already labeled "X", then Si, Sj is labeled "X".**
4. **Continue executing Step 3 until no new squares are marked with "X".**
5. **For each remaining unmarked square Si, Sj, then Si and Sj are equivalent.**

State Assignment

When FSM implemented with gate logic, number of gates will depend on mapping between symbolic state names and binary encodings

4 states = 4 choices for first state, 3 for second, 2 for third, 1 for last
= 24 different encodings (4!)

Example for State Assignment: Traffic Light Controller

HG	HY	FG	FY
00	01	10	11
00	01	11	10
00	10	01	11
00	10	11	01
00	11	01	10
00	11	10	01
01	00	10	11
01	00	11	10
01	10	00	11
01	10	11	00
01	11	00	10
01	11	10	00

HG	HY	FG	FY
10	00	01	11
10	00	11	01
10	01	00	11
10	01	11	00
10	11	00	01
10	11	01	00
11	00	01	10
11	00	10	01
11	01	00	10
11	01	10	00
11	10	00	01
11	10	01	00

Inputs			Present State	Next State	Outputs		
C	TL	TS	Q_1 Q_0	P_1 P_0	ST	H_1 H_0	F_1 F_0
0	X	X	HG	HG	0	00	10
X	0	X	HG	HG	0	00	10
1	1	X	HG	HY	1	00	10
X	X	0	HY	HY	0	01	10
X	X	1	HY	FG	1	01	10
1	0	X	FG	FG	0	10	00
0	X	X	FG	FY	1	10	00
X	1	X	FG	FY	1	10	00
X	X	0	FY	FY	0	10	01
X	X	1	FY	HG	1	10	01

24 state assignments for the traffic light controller

Symbolic State Names: HG, HY, FG, FY

State Assignment

Some Sample Assignments for the Traffic Light Controller

Espresso input format:

```
.i 5
.o 7
.ilb c tl ts q1 q0
.ob p1 p0 st h1 h0 f1 f0
.p 10
0--   HG  HG 00010
-0-   HG  HG 00010
11-  HG  HY 10010
--0   HY  HY 00110
--1   HY  FG 10110
10-  FG  FG  01000
0--   FG  FY 11000
-1-   FG  FY 11000
--0   HY  FY  01001
--1   HY  HG 11001
.e
```

Two assignments: HG = 00, HY = 01, FG = 11, FY = 10
HG = 00, HY = 10, FG = 01, FY = 11

State Assignment

Random Assignments Evaluated with Espresso

```
.i 5
.o 7
.ilb c tl ts q1 q0
.ob p1 p0 st h1 h0 f1 f0
.p 9
11-00  0110000
10-11  1101000
--101   1010000
--010   1001001
---01    0100100
--110   0011001
---0-     0000010
0--11   1011000
-1-11   1011000
.e
```

26 literals
9 unique terms
several 5 and 4 input gates
13 gates total

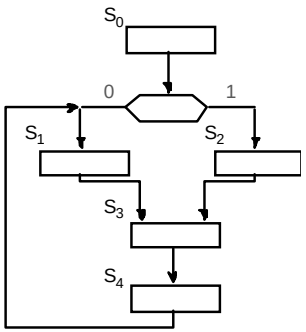
```
.i 5
.o 7
.ilb c tl ts q1 q0
.ob p1 p0 st h1 h0 f1 f0
.p 8
11-0-  1010000
--010  1000100
0--01  1010000
--110  0110100
--111  0011001
----0    0000010
---01   0101000
--011  1101001
.e
```

21 literals
8 unique terms
no 5 input gates, 2 4 input gates
14 gates total

State Assignment

Pencil & Paper Heuristic Methods

State Maps: similar in concept to K-maps
If state X transitions to state Y, then assign "close" assignments to X and Y

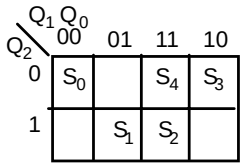


State Name	Assignment Q_2	Q_1	Q_0
S_0	0	0	0
S_1	1	0	1
S_2	1	1	1
S_3	0	1	0
S_4	0	1	1

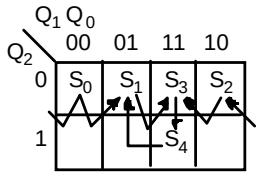
Assignment

State Name	Assignment Q_2	Q_1	Q_0
S_0	0	0	0
S_1	0	0	1
S_2	0	1	0
S_3	0	1	1
S_4	1	1	1

Assignment



State Map



State Map

State Assignment

Paper and Pencil Methods

Minimum Bit Distance Criterion

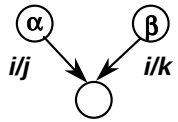
Transition	*First Assignment Bit Changes*	*Second Assignment Bit Changes*
S0 to S1:	2	1
S0 to S2:	3	1
S1 to S3:	3	1
S2 to S3:	2	1
S3 to S4:	1	1
S4 to S1:	2	2
	13	7

Traffic light controller: HG = 00, HY = 01, FG = 11, FY = 10
yields minimum distance encoding but not best assignment!

State Assignment

Paper & Pencil Methods

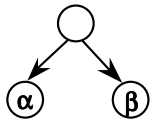
Alternative heuristics based on input and output behavior as well as transitions:



Highest Priority

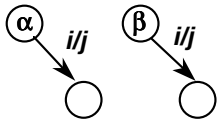
Adjacent assignments to:

states that share a common next state (group 1's in next state map)



Medium Priority

states that share a common ancestor state (group 1's in next state map)



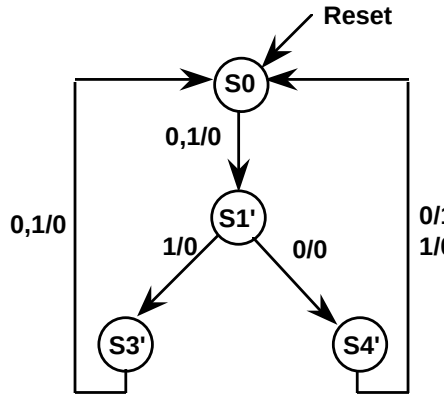
Lowest Priority

states that have common output behavior (group 1's in output map)

State Assignment

Pencil and Paper Methods

Example: 3-bit Sequence Detector



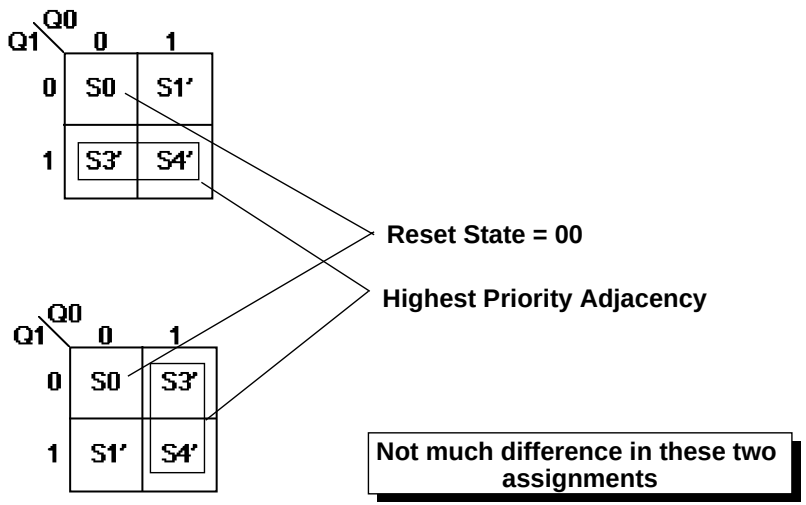
Highest Priority: (S3', S4')

Medium Priority: (S3', S4')

Lowest Priority:
0/0: (S0, S1', S3')
1/0: (S0, S1', S3', S4')

State Assignment

Paper and Pencil Methods

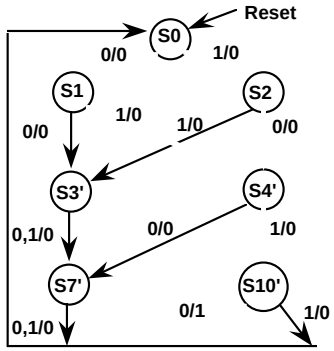


Not much difference in these two assignments

State Assignment

Paper & Pencil Methods

Another Example: 4 bit String Recognizer



Highest Priority: (S3', S4'), (S7', S10')

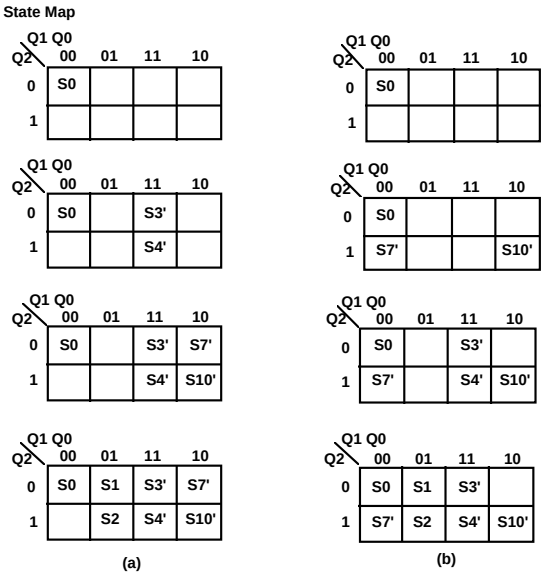
Medium Priority:
(S1, S2), 2x(S3', S4'), (S7', S10')

Lowest Priority:
0/0: (S0, S1, S2, S3', S4', S7')
1/0: (S0, S1, S2, S3', S4', S7')

State Assignment

Paper & Pencil Methods

State Map



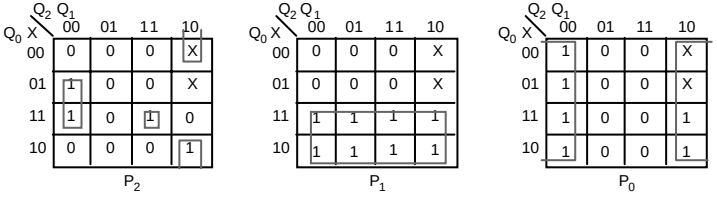
00 = Reset = S0

(S1, S2), (S3', S4'), (S7', S10') placed adjacently

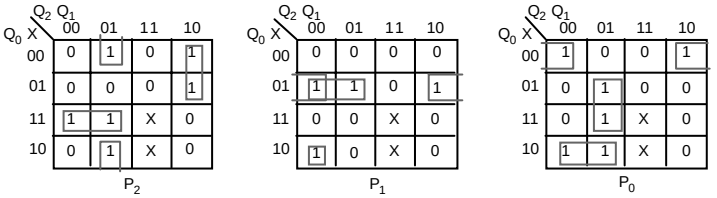
State Assignment

Effect of Adjacencies on Next State Map

Current State	Next State X = 0	Next State X = 1
(S_0) 000	001	101
(S_1) 001	011	111
(S_2) 101	111	011
(S'_3) 011	010	010
(S'_4) 111	010	110
(S'_7) 010	000	000
(S'_{10}) 110	000	000



Current State	Next State X = 0	Next State X = 1
(S_0) 000	001	010
(S_1) 001	011	100
(S_2) 010	100	011
(S'_3) 011	101	101
(S'_4) 100	101	110
(S'_7) 101	000	000
(S'_{10}) 110	000	000



First encoding exhibits a better clustering of 1's in the next state map

State Assignment

One Hot Encodings

n states encoded in n flipflops

HG = 0001
HY = 0010
FG = 0100
FY = 1000

Complex logic for discrete gate implementation

```
.i 7
.o 9
.ilb c tl ts q3 q2 q1 q0
.ob p3 p2 p1 p0 st h1 h0 f1 f0
.p 10
0-- 0001 0001 00010
-0- 0001 0001 00010
11- 0001 0010 10010
--0 0010 0010 00110
--1 0010 0100 10110
10- 0100 0100 01000
0-- 0100 1000 11000
-1- 0100 1000 11000
--0 0010 1000 01001
--1 0010 0001 11001
.e
```

Espresso Inputs

```
.i 7
.o 9
.ilb c tl ts q3 q2 q1 q0
.ob p3 p2 p1 p0 st h1 h0 f1 f0
.p 8
10-0100 010001000
11-0001 001010010
-0-0001 000100010
0--0001 000100010
0--0100 100011000
-1-0100 100011000
--00010 101001111
--10010 010111111
.e
```

Espresso Outputs

State Assignment

Computer-Based Methods

NOVA: State Assignment for 2-Level Circuit Networks

Input Constraints: states with same i/o mapped to same next state

Output Constraints: states which are successors of same predecessor

NOVA Input File for the Traffic Light Controller

inputs current_state next_state outputs

```
0--  HG HG  00010
-0-  HG HG  00010
11-  HG HY  10010
--0  HY HY  00110
--1  HY FG  10110
10-  FG FG  01000
0--  FG FY  11000
-1-  FG FY  11000
--0  FY FY  01001
--1  FY HG  11001
```

```
nova -e <encoding strategy> -t <nova input file>
```


State Assignment

Computer-Based Methods

Greedy: satisfy as many input constraints as possible

Hybrid: satisfy input constraints, more sophisticated improvement strategy

I/O Hybrid: satisfy both input and output constraints

Exact: satisfy ALL input conditions

Input Annealing: like hybrid, but uses an even improvement strategy

1-Hot: uses a 1-hot encoding

Random: uses a randomly generated encoding

State Assignment

Computer-Based Methods

	HG	HY	FG	FY	# of Product Terms
Greedy (-e ig):	00	11	01	10	9
Hybrid (-e ih):	00	11	10	01	9
Exact (-e ie):	11	10	01	00	10
IO Hybrid (-e ioh):	00	01	11	10	9
I Annealing (-e ia):	01	10	11	10	9
Random (-e r):	11	00	01	10	9

-z HG **on the command line forces NOVA to assign 00 to state HG**

State Assignment

Computer Based Methods

More NOVA Examples:

3-bit Recognizer

```
- S0 S1 0
0 S1 S3 0
1 S1 S4 0
- S3 S0 0
0 S4 S0 1
1 S4 S0 0
```

	S0	S1'	S3'	S4'	terms
Greedy:	00	01	11	10	4
Hybrid:	00	01	10	11	4
Exact:	00	01	10	11	4
IO Hyb:	00	10	01	11	4
I Ann:	00	01	11	10	4
Random:	00	11	10	01	4

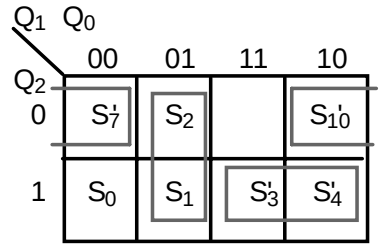
State Assignment

Computer Based Methods

More NOVA Examples:

4-bit Recognizer

```
0  S0  S1  0
1  S0  S2  0
0  S1  S3  0
1  S1  S4  0
0  S2  S4  0
1  S2  S3  0
-  S3  S7  0
0  S4  S7  0
1  S4  S7  0
-  S7  S0  0
0  S10 S0  1
1  S10 S0  0
```



Same adjacencies as the heuristic method

	S0	S1	S2	S3'	S4'	S7'	S10'	terms
Greedy:	100	110	010	011	111	000	001	7
Hybrid:	101	110	111	001	011	000	010	7
Exact:	101	110	111	001	011	000	010	7
IO Hyb:	110	011	001	100	101	000	010	7
I Ann:	100	101	001	111	110	000	010	6
Rand:	011	100	101	110	111	000	001	7

State Assignment

Mustang

Oriented towards multi-level logic implementation

Goal is to reduce literal count, rather than product terms

Input Format:

```
.i 3          # inputs
.o 5          # outputs
.s 2          # of state bits
0--  HG HG  00010
-0-  HG HG  00010
11-  HG HY  10010
--0  HY HY  00110
--1  HY FG  10110
10-  FG FG  01000
0--  FG FY  11000
-1-  FG FY  11000
--0  FY FY  01001
--1  FY HG  11001
```


State Assignment

Mustang

Goal: maximum common subexpressions in next state function

Encoding Strategies:

Random

Sequential

One-Hot

Fan-in: **states with common predecessors given adjacent assignment**

Fan-out: **state with common next state and outputs given adjacent assignments**

State Assignment

Mustang

Traffic Light:

	HG	HY	FG	FY	# product terms
Random:	01	10	11	00	9
Sequential:	01	10	11	00	9
Fan-in:	00	01	10	11	8
Fan-out:	10	11	00	01	8

3 Bit String Recognizer:

	S0	S1	S3'	S4'	# product terms
Random:	01	10	11	00	5
Sequential:	01	10	11	00	5
Fan-in:	10	11	00	01	4
Fan-out:	10	11	00	01	4

4 Bit String Recognizer:

	S0	S1	S2	S3'	S4'	S7'	S10'	# product terms
Random:	101	010	011	110	111	001	000	8
Sequential:	001	010	011	100	101	110	111	8
Fan-in:	100	010	011	000	001	101	110	8
Fan-out:	110	010	011	100	101	000	001	6

To obtain multilevel implementations, must run misII

State Assignment

JEDI

Input Format:

# of states; encoding bit length

```
.i 4
.o 4
.enum States 4 2 HG HY FG FY
.enum Colors 3 2 GREEN RED YELLOW
0 - - HG HG 0 GREEN RED
- 0 - HG HG 0 GREEN RED
1 1 - HG HY 1 GREEN RED
- - 0 HY HY 0 YELLOW RED
- - 1 HY FG 1 YELLOW RED
1 0 - FG FG 0 RED GREEN
0 - - FG FY 1 RED GREEN
- 1 - FG FY 1 RED GREEN
- - 0 FY FY 0 RED YELLOW
- - 1 FY HG 1 RED YELLOW
```

Kinds of states

Kinds of outputs

General encoding problems: best encodings for states and outputs

Encoding Strategies:

- Random
- Straightforward
- One Hot
- Input Dominant
- Output Dominant
- Modified Output Dominate
- I/O Combination

State Assignment

JEDI

Traffic Light:

	HG	HY	FG	FY	Grn	Yel	Red	# product terms
Input:	00	10	11	01	11	01	00	9
Output:	00	01	11	10	10	11	01	9
Comb:	00	10	11	01	10	00	01	9
Output':	01	00	10	11	10	00	01	10

3 Bit String Recognizer:

	S0	S1	S3'	S4'	# product terms
Input:	01	10	11	00	4
Output:	01	10	11	00	4
Comb.:	10	11	00	01	4
Output':	10	11	00	01	4

4 Bit String Recognizer:

	S0	S1	S2	S3'	S4'	S7'	S10'	# product terms
Input:	111	101	100	010	110	011	001	7
Output:	101	110	100	010	000	111	011	7
Comb.:	100	011	111	110	010	000	101	7
Output':	001	100	101	010	011	000	111	8

To obtain multilevel implementations, must run misII

State Assignment

Mustang vs. Jedi

Traffic Light Controller

Mustang

```
Q1 = HL0 " TS + FL0 " TS' + FL0' " P1
Q0 = Q1 " C' " P1 + C " TL " P0' + TS' " P0
ST = Q0 " P0' + Q0' " P0
HL1 = FL0 + P1 " P0'
HL0 = P1' " P0
FL1 = P1'
FL0 = HL0' " P0
```

Fewer wires

Jedi

```
Q1 = HL1 " C " TL + HL0 + FL1 " C " TL'
Q0 = HL0 " TS + FL1 + FL0 " TS'
ST = Q1 " HL1 + Q1' " FL0 + HL1' " FL1' " TS
HL1 = P1' " P0'
HL0 = P1 " P0'
FL1 = HL0' " P1
FL0 = HL1' " P1'
```

Fewer literals

Choice of Flipflops

J-K FFs: reduce gate count, increase # of connections

D FFs: simplify implementation process, decrease # of connections

Procedure:

1. **Given state assignments, derive the next state maps from the state transition table**
2. **Remap the next state maps given excitation tables for a given FF**
3. **Minimize the remapped next state function**

Choice of Flipflops

Examples

4 bit Sequence Detector using NOVA derived state assignment

Encoded State Transition Table

Present State	Next State I=0	Next State I=1	Output I=0	Output I=1
000 (S_0)	011 (S_1)	010 (S_2)	0	0
011 (S_1)	101 (S_3')	111 (S_4')	0	0
010 (S_2)	111 (S_4')	101 (S_3')	0	0
101 (S_3')	100	100 (S_7')	0	0
111 (S_4')	(S_7')	110	0	0
100 (S_7')	100	(S_{10}')	0	0
110 (S_{10}')	(S_7')	000	1	0

Encoded Next State Map

Present State	Next State I=0	Next State I=1
000	011	010
001	XXX	XXX
010	111	101
011	101	111
100	000	000
101	100	100
110	000	000
111	100	110

Choice of Flipflops

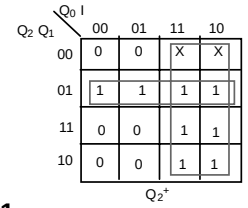
D FF Implementation

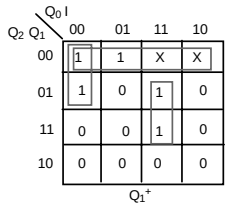
$$D_{Q2+} = \overline{Q2}\,"\,Q1 + Q0$$

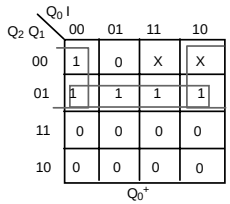
$$D_{Q1+} = Q1\,"\,Q0\,"\,I + \overline{Q2}\,"\,\overline{Q0}\,"\,\overline{I} + Q2\,"\,Q1$$

$$D_{Q0+} = \overline{Q2}\,"\,Q1 + \overline{Q2}\,"\,\overline{I}$$

6 product terms
15 literals







Choice of Flipflops

J-K Implementation

Present State	Next State I=0	Next State I=1	Remapped Next State J I=0	Remapped Next State K I=0	Remapped Next State J I=1	Remapped Next State K I=1
000	011	010	011	XXX	010	XXX
001	XXX	XXX	XXX	XXX	XXX	XXX
010	111	101	1X1	X0X	1X1	X1X
011	101	111	1XX	X10	1XX	X00
100	000	000	X00	1XX	X00	1XX
101	100	100	X0X	0X1	X0X	0X1
110	000	000	XX0	11X	XX0	11X
111	100	110	XXX	011	XXX	001

Remapped Next State Table

Choice of Flipflops

J-K Implementation (continued)

$J_{Q2+} = Q1$

$K_{Q2+} = \overline{Q0}$

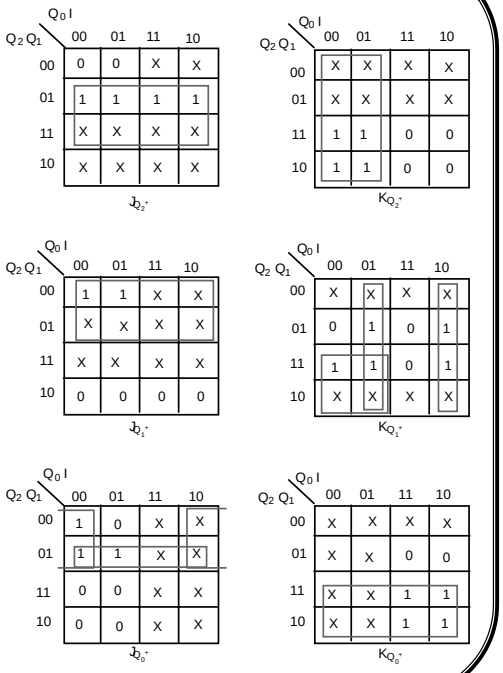
$J_{Q1+} = \overline{Q2}$

$K_{Q1+} = \overline{Q0} \cdot I + Q0 \cdot \overline{I} + Q2 \cdot \overline{Q0}$

$J_{Q0+} = \overline{Q2} \cdot Q1 + \overline{Q2} \cdot \overline{I}$

$K_{Q0+} = Q2$

9 unique terms
14 literals

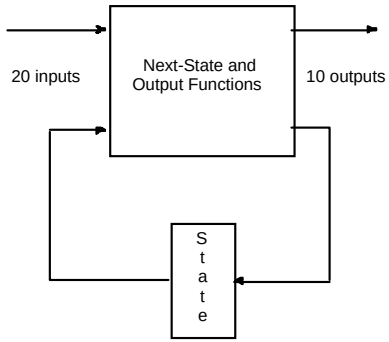


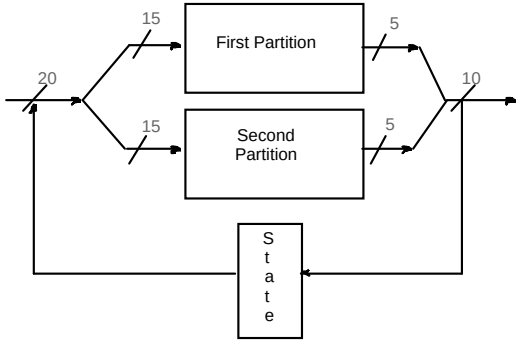
Finite State Machine Partitioning

Why Partition?

mapping FSMs onto programmable logic components:

- **limited number of input/output pins**
- **limited number of product terms or other programmable resources**



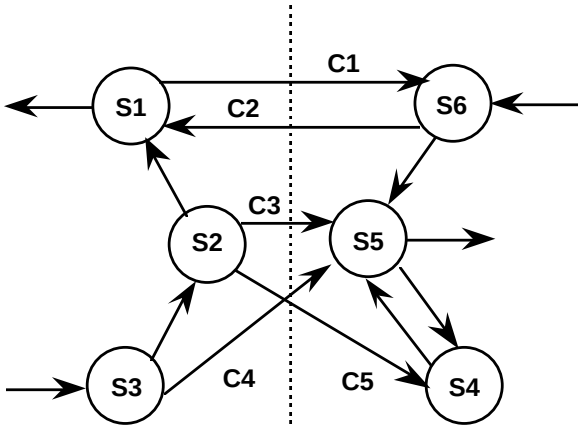


Example of Input/Output Partitioning:
5 outputs depend on 15 inputs
5 outputs depend on different overlapping set of 15 inputs

Finite State Machine Partitioning

Introduction of Idle States

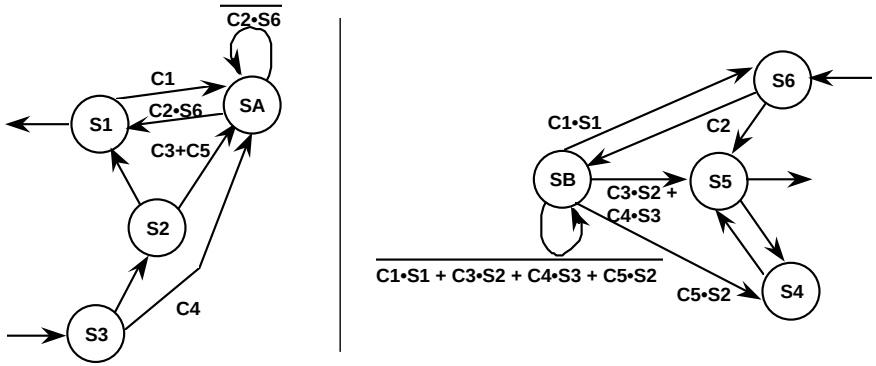
Before Partitioning



Finite State Machine Partitioning

Introduction of Idle States

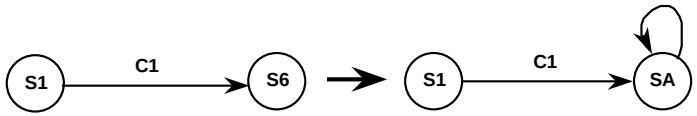
After Partitioning



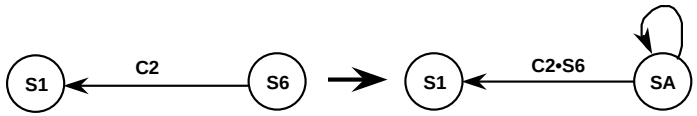
Finite State Machine Partitioning

Rules for Partitioning

Rule #1: Source State Transformation; SA is the Idle State



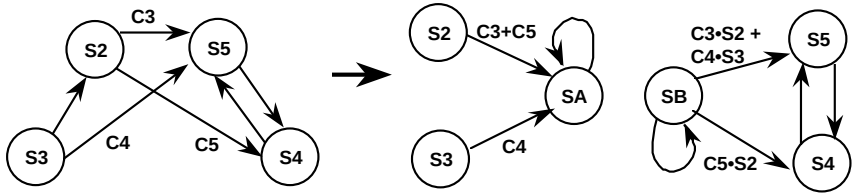
Rule #2: Destination State Transformation



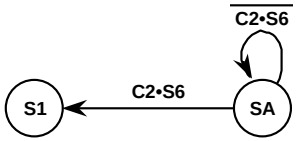
Finite State Machine Partitioning

Rules for Partitioning

Rule #3: Multiple Transitions with Same Source or Destination

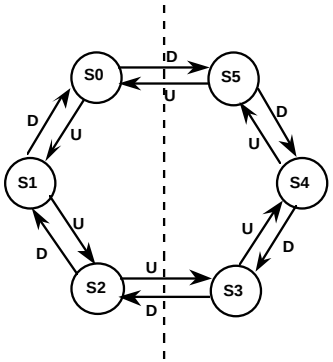


Rule #4: Hold Condition for Idle State



Finite State Machine Partitioning

Another Example

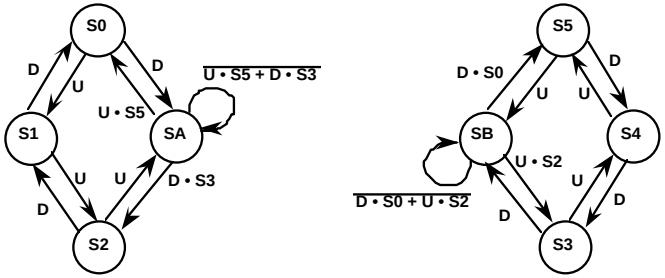


6 state up/down counter

building block has 2 FFs + combinational logic

Finite State Machine Partitioning

6 State Up/Down Counter



Introduction of the two idle state SA, SB

Count sequence S0, S1, S2, S3, S4, S5:
S2 goes to SA and holds, leaves after S5
S5 goes to SB and holds, leaves after S2

Down sequence is similar

Chapter Summary

" ***Optimization of FSM***

State Reduction Methods: Row Matching, Implication Chart

State Assignment Methods: Heuristics and Computer Tools

" ***Implementation Issues***

Choice of Flipflops

Finite State Machine Partitioning

Chapter #10: Finite State Machine Implementation

Contemporary Logic Design

Randy H. Katz
University of California, Berkeley

July 1993

Chapter Outline

" *Implementation Strategies*

discrete logic

design with counters, ROMs

programmable logic

PALs

FGPAs: Altera, Actel, Xilinx

Implementation Strategies

" Discrete Gate Logic

Emphasis so far

" MSI Logic (e.g., Counters)

" Structured Logic (e.g., PLA/PAL, ROM)

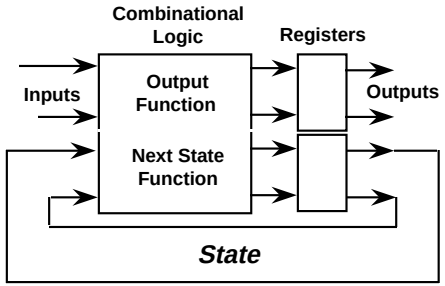
" Field Programmable Gate Arrays (FPGAs)

Function can be configured "on the fly" or in the field

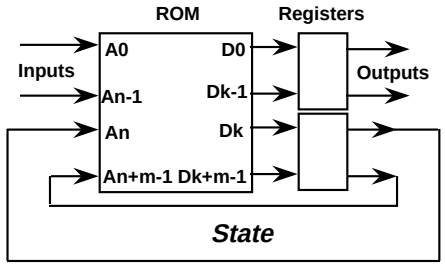
Flipflops/Registers plus discrete gates on the same chip

Implementation Strategies

FSM Design with Structured Logic



Block Diagram for Synchronous Mealy Machine



ROM-based Realization

" Inputs & Current State form the address

" ROM data bits form the Outputs & Next State

Implementation Strategies

ROM-based Design

Example: BCD to Excess 3 Serial Converter

BCD	Excess 3 Code
0000	0011
0001	0100
0010	0101
0011	0110
0100	0111
0101	1000
0110	1001
0111	1010
1000	1011
1001	1100

Conversion Process

Bits are presented in bit serial fashion starting with the least significant bit

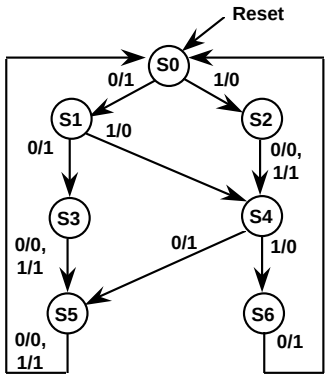
Single input X, single output Z

Implementation Strategies

BCD to Excess-3 Converter

Present State	Next State X=0	Next State X=1	Output X=0	Output X=1
S0	S1	S2	1	0
S1	S3	S4	1	0
S2	S4	S4	0	1
S3	S5	S5	0	1
S4	S5	S6	1	0
S5	S0	S0	0	1
S6	S0	--	1	--

State Transition Table



Derived State Diagram

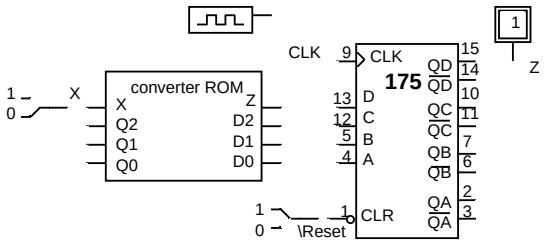
Implementation Strategies

BCD to Excess 3 Converter

ROM-based Implementation

ROM Address X	Q2	Q1	Q0	ROM Outputs Z	D2	D1	D0
0	0	0	0	1	0	0	1
0	0	0	1	1	0	1	1
0	0	1	0	0	1	0	0
0	0	1	1	0	1	0	1
0	1	0	0	1	1	0	1
0	1	0	1	0	0	0	0
0	1	1	0	1	0	0	0
0	1	1	1	X	X	X	X
1	0	0	0	0	0	1	0
1	0	0	1	0	1	0	0
1	0	1	0	1	1	0	0
1	0	1	1	1	1	0	1
1	1	0	0	0	1	1	0
1	1	0	1	1	0	0	0
1	1	1	0	X	X	X	X
1	1	1	1	X	X	X	X

Truth Table/ROM I/Os



Circuit Level Realization
74175 = 4 x positive edge triggered D FFs

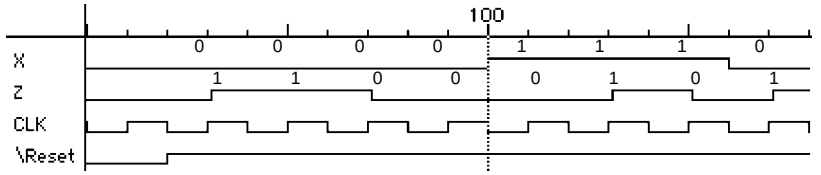
In ROM-based designs, no need to consider state assignment

Implementation Strategies

BCD to Excess-3 Converter

LSB MSB

Timing Behavior for input strings 0 0 0 0 (0) and 1 1 1 0 (7)



Implementation Strategies

BCD to Excess 3 Converter

PLA-based Design

State Assignment with NOVA

```
0 S0 S1 1
1 S0 S2 0
0 S1 S3 1
1 S1 S4 0
0 S2 S4 0
1 S2 S4 1
0 S3 S5 0
1 S3 S5 1
0 S4 S5 1
1 S4 S6 0
0 S5 S0 0
1 S5 S0 1
0 S6 S0 1
```

NOVA input file

S0 = 000
S1 = 001
S2 = 011
S3 = 110
S4 = 100
S5 = 111
S6 = 101

NOVA derived state assignment

9 product term implementation

Implementation Strategies

BCD to Excess 3 Converter

Espresso Inputs

```
.i 4
.o 4
.ilb x q2 q1 q0
.ob d2 d1 d0 z
.p 16
0 000 001 1
1 000 011 0
0 001 110 1
1 001 100 0
0 011 100 0
1 011 100 1
0 110 111 0
1 110 111 1
0 100 111 1
1 100 101 0
0 111 000 0
1 111 000 1
0 101 000 1
1 101 --- -
0 010 --- -
1 010 --- -
.e
```

Espresso Outputs

```
.i 4
.o 4
.ilb x q2 q1 q0
.ob d2 d1 d0 z
.p 9
0001 0100
10-0 0100
01-0 0100
1-1- 0001
-0-1 1000
0-0- 0001
-1-0 1000
--10 0100
---0 0010
.e
```


Implementation Strategies

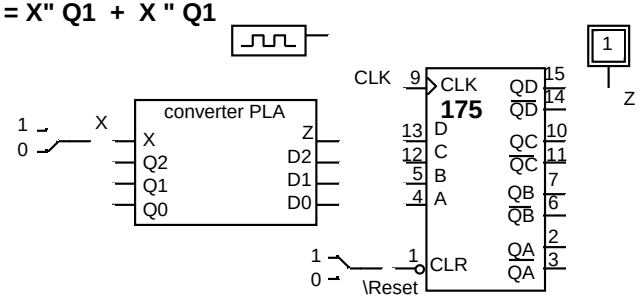
BCD to Excess 3 Converter

D2 = $\overline{Q2}$ " Q0 + Q2 " $\overline{Q0}$

D1 = $\overline{X}$ " $\overline{Q2}$ " $\overline{Q1}$ " Q0 + X " $\overline{Q2}$ " Q0 + $\overline{X}$ " Q2 " $\overline{Q0}$ + Q1 " $\overline{Q0}$

D0 = $\overline{Q0}$

Z = X" Q1 + $\overline{X}$ " $\overline{Q1}$



Implementation Strategies

BCD to Excess 3 Serial Converter

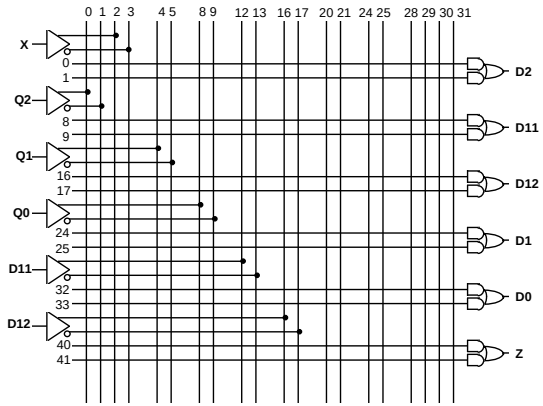
10H8 PAL: 10 inputs, 8 outputs, 2 product terms per OR gate

D1 = D11 + D12

D11 = $\overline{X}$ " $\overline{Q2}$ " $\overline{Q1}$ " Q0 + X " $\overline{Q2}$ " $\overline{Q0}$

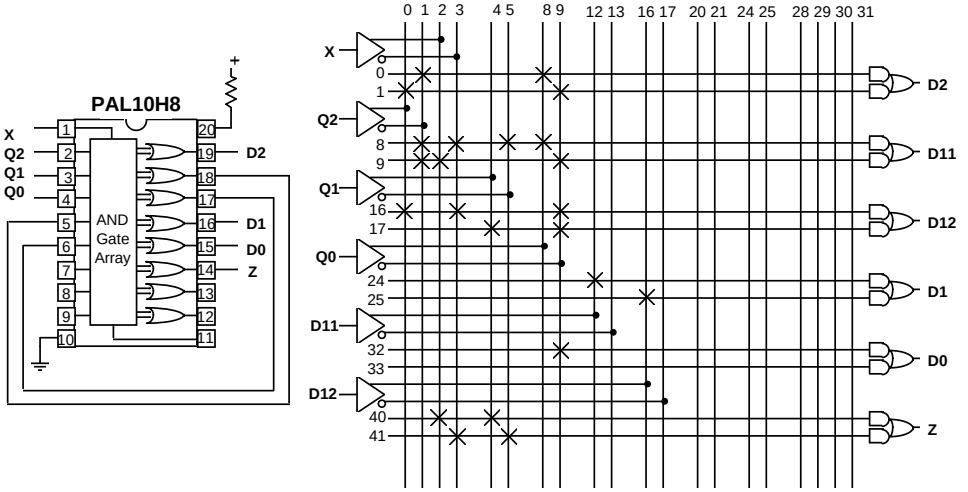
D12 = $\overline{X}$ " Q2 " $\overline{Q0}$ + Q1 " $\overline{Q0}$

0. $\overline{Q2}$ " Q0
1. Q2 " $\overline{Q0}$
8. X " Q2 " Q1 " Q0
9. X " Q2 " Q0
16. X " $\overline{Q2}$ " Q0
17. Q1 " Q0
24. D11
25. D12
32. Q0
33. not used
40. X " Q1
41. X " Q1



Implementation Strategies

BCD to Excess 3 Serial Converter

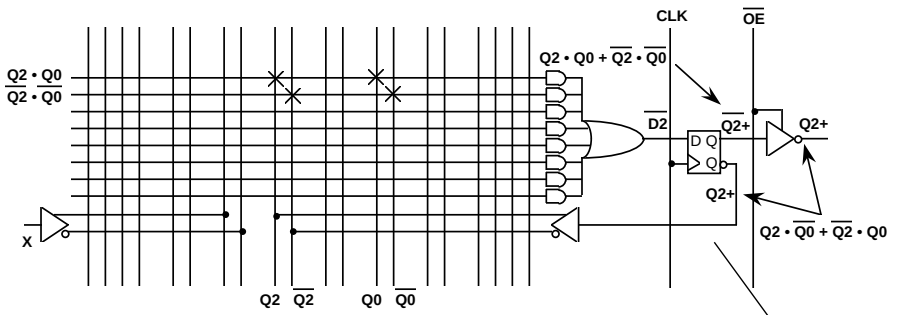


Implementation Strategies

More Advanced PAL Architectures

Registered PAL Architecture

Buffered Input or product term



Negative Logic Feedback

$\overline{D2} = Q2 \cdot Q0 + \overline{Q2} \cdot \overline{Q0}$

$\overline{D1} = \overline{X} \cdot \overline{Q2} \cdot \overline{Q1} \cdot \overline{Q0} + X \cdot Q2 + X \cdot Q0 + Q2 \cdot Q0 + Q1 \cdot Q0$

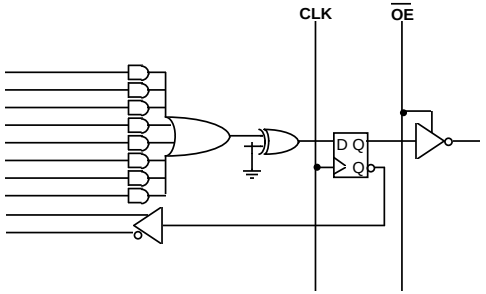
$\overline{D0} = Q0$

$\overline{Z} = X \cdot \overline{Q1} + \overline{X} \cdot Q1$

Implementation Strategies

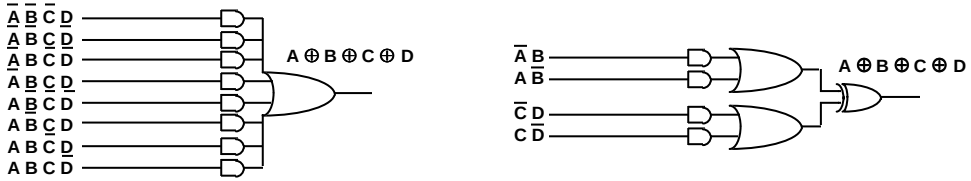
Advanced PAL Architectures

Programmable Output Polarity/XOR PALs



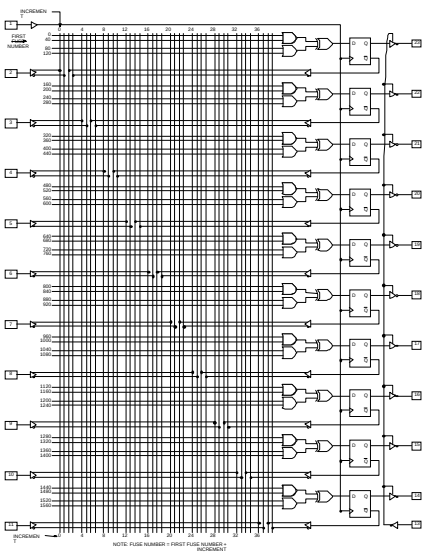
Buried Registers: decouple FF from the output pin

Advantage of XOR PALs: Parity and Arithmetic Operations

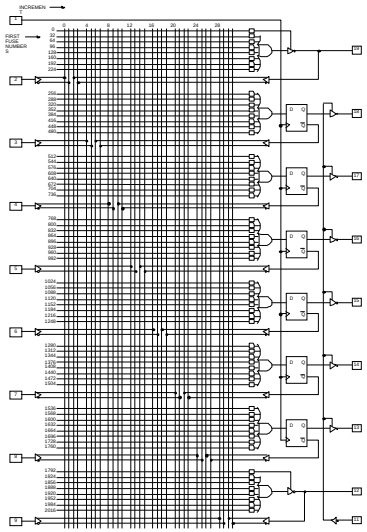


Implementation Strategies

Example of XOR PAL



Example of Registered PAL



Implementation Strategies

Specifying PALs with ABEL

P10H8 PAL

```
module bcd2excess3
title 'BCD to Excess 3 Code Converter State Machine'
u1 device 'p10h8';

"Input Pins
 X,Q2,Q1,Q0,D11i,D12i  pin  1,2,3,4,5,6;

"Output Pins
 D2,D11o,D12o,D1,D0,Z  pin  19,18,17,16,15,14;

INSTATE = [Q2, Q1, Q0];
S0 = [0, 0, 0];
S1 = [0, 0, 1];
S2 = [0, 1, 1];
S3 = [1, 1, 0];
S4 = [1, 0, 0];
S5 = [1, 1, 1];
S6 = [1, 0, 1];

equations
 D2 = (!Q2 & Q0) # (Q2 & !Q0);
 D1 = D11i # D12i;
 D11o = (!X & !Q2 & !Q1 & Q0) # (X & !Q2 & !Q0);
 D12o = (!X & Q2 & !Q0) # (Q1 & !Q0);
 D0 = !Q0;
 Z = (X & Q1) # (!X & !Q1);

end bcd2excess3;
```

Explicit equations for partitioned output functions

Implementation Strategies

Specifying PALs with ABEL

P12H6 PAL

```
module bcd2excess3
title 'BCD to Excess 3 Code Converter State Machine'
u1 device  'p12h6';

"Input Pins
 X, Q2, Q1, Q0  pin  1, 2, 3, 4;

"Output Pins
 D2, D1, D0, Z  pin  17, 18, 16, 15;

INSTATE = [Q2, Q1, Q0]; OUTSTATE = [D2, D1, D0];
S0in = [0, 0, 0];  S0out = [0, 0, 0];
S1in = [0, 0, 1];  S1out = [0, 0, 1];
S2in = [0, 1, 1];  S2out = [0, 1, 1];
S3in = [1, 1, 0];  S3out = [1, 1, 0];
S4in = [1, 0, 0];  S4out = [1, 0, 0];
S5in = [1, 1, 1];  S5out = [1, 1, 1];
S6in = [1, 0, 1];  S6out = [1, 0, 1];

equations
 D2 = (!Q2 & Q0) # (Q2 & !Q0);
 D1 = (!X & !Q2 & !Q1 & Q0) # (X & !Q2 & !Q0) #
   (!X & Q2 & !Q0) # (Q1 & !Q0);
 D0 = !Q0;
 Z = (X & Q1) # (!X & !Q1);

end bcd2excess3;
```

Simpler equations

Implementation Strategies

Specifying PALs with ABEL

P16R4 PAL

```
module bcd2excess3
title 'BCD to Excess 3 Code Converter'
u1 device 'p16r4';

"Input Pins
 Clk, Reset, X, !OE     pin  1, 2, 3, 11;

"Output Pins
 D2, D1, D0, Z     pin 14, 15, 16, 13;

SREG = [D2, D1, D0];
S0 = [0, 0, 0];
S1 = [0, 0, 1];
S2 = [0, 1, 1];
S3 = [1, 1, 0];
S4 = [1, 0, 0];
S5 = [1, 1, 1];
S6 = [1, 0, 1];
```

```
state_diagram SREG
state S0: if Reset then S0
     else if X then S2 with Z = 0
     else S1 with Z = 1
state S1: if Reset then S0
     else if X then S4 with Z = 0
     else S3 with Z = 1
state S2: if Reset then S0
     else if X then S4 with Z = 1
     else S4 with Z = 0
state S3: if Reset then S0
     else if X then S5 with Z = 1
     else S5 with Z = 0
state S4: if Reset then S0
     else if X then S6 with Z = 0
     else S5 with Z = 1
state S5: if Reset then S0
     else if X then S0 with Z = 1
     else S0 with Z = 0
state S6: if Reset then S0
     else if !X then S0 with Z = 1

end bcd2excess3;
```


Implementation Strategies

FSM Design with Counters

Synchronous Counters: CLR, LD, CNT

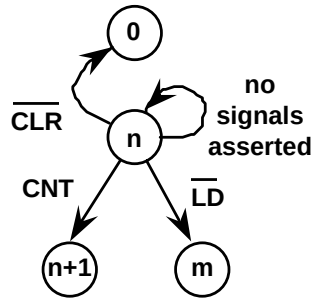
Four kinds of transitions for each state:

(1) to State 0 (CLR)

(2) to next state in sequence (CNT)

(3) to arbitrary next state (LD)

(4) loop in current state

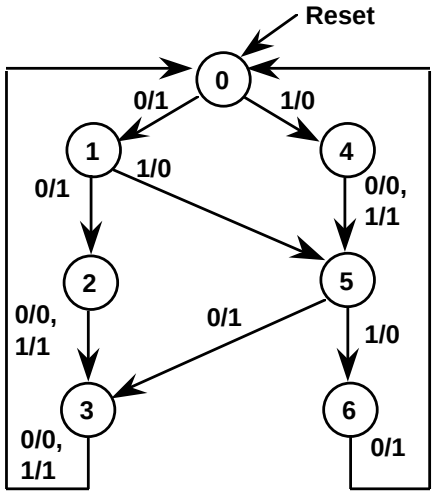


Careful state assignment is needed to reflect basic sequencing of the counter

Implementation Strategies

FSM Design with Counters

Excess 3 Converter Revisited



Note the sequential nature of the state assignments

Implementation Strategies

FSM Design with Counters

Excess 3 Converter

Inputs/Current State				*Next State*			*Outputs*						
X	Q2	Q1	Q0	Q2+	Q1+	Q0+	Z	$\overline{CLR}$	$\overline{LD}$	EN	C	B	A
0	0	0	0	0	0	1	1	1	1	1	X	X	X
0	0	0	1	0	1	0	1	1	1	1	X	X	X
0	0	1	0	0	1	1	0	1	1	1	X	X	X
0	0	1	1	0	0	0	0	0	X	X	X	X	X
0	1	0	0	1	0	1	1	1	1	1	X	X	X
0	1	0	1	0	1	1	0	1	0	X	0	1	0
0	1	1	0	0	0	0	1	0	X	X	X	X	X
0	1	1	1	X	X	X	X	X	X	X	X	X	X
1	0	0	0	1	0	0	0	1	0	X	1	0	0
1	0	0	1	1	0	1	0	1	0	X	1	0	1
1	0	1	0	0	1	1	1	1	1	1	X	X	X
1	0	1	1	0	0	0	1	0	X	X	X	X	X
1	1	0	0	1	0	1	0	1	1	1	X	X	X
1	1	0	1	1	1	0	1	1	1	1	X	X	X
1	1	1	0	X	X	X	X	X	X	X	X	X	X
1	1	1	1	X	X	X	X	X	X	X	X	X	X

CLR signal dominates LD which dominates Count

Implementation Strategies

Implementing FSMs with Counters

Excess 3 Converter

Espresso Input File

```
.i 5
.o 7
.ilb res x q2 q1 q0
.ob z clr ld en c b a
.p 17
1---- -0-----
00000 1111---
00001 1111---
00010 0111---
00011 00-----
00100 0111---
00101 110-011
00110 10-----
00111 -------
01000 010-100
01001 010-101
01010 1111---
01011 10-----
01100 1111---
01101 0111---
01110 -------
01111 -------
.e
```

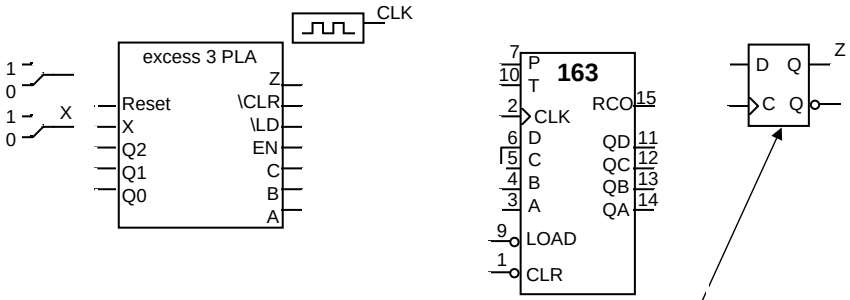
Espresso Output File

```
.i 5
.o 7
.ilb res x q2 q1 q0
.ob z clr ld en c b a
.p 10
0-001 0101101
-0-01 1000000
-11-0 1000000
0-0-0 0101100
-000- 1010000
-0--0 0010000
0-10- 0101011
--11- 1000000
-11-- 0010000
-1-1- 1010000
.e
```


Implementation Strategies

FSM Implementation with Counters

Excess 3 Converter Schematic



Synchronous Output Register

Implementation Strategies

FSM Design with More Sophisticated PLDs

Programmable Logic Devices = PLD

PALs, PLAs = 10 - 100 Gate Equivalents

Field Programmable Gate Arrays = FPGAs

" Altera MAX Family

" Actel Programmable Gate Array

" Xilinx Logical Cell Array

100 - 1000(s) of Gate Equivalents!

Implementation Strategies

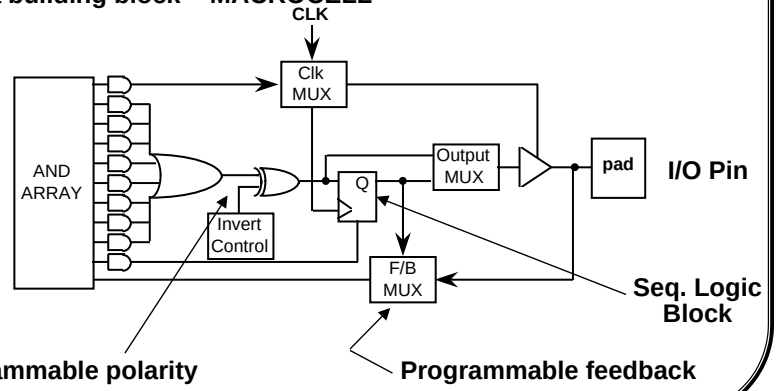
Design with More Sophisticated PLDs

Altera EPLD (Erasable Programmable Logic Devices)

Historical Perspective:
PALs – same technology as programmed once bipolar PROM
EPLDs — CMOS erasable programmable ROM (EPROM)
erased by UV light

Altera building block = MACROCELL

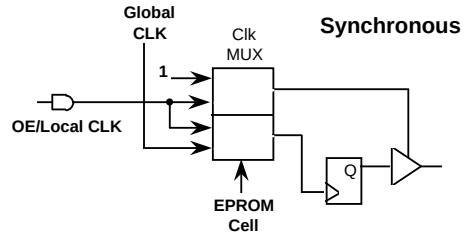
8 Product Term AND-OR Array + Programmable MUX's



Implementation Strategies

Design with More Sophisticated PLDs

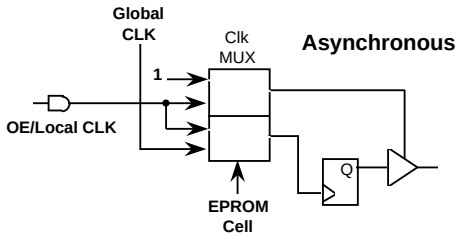
Altera EPLDs contain 8 to 48 independently programmed macrocells
Personalized by EPROM bits:



Synchronous Mode

Flipflop controlled by global clock signal

local signal computes output enable



Asynchronous Mode

Flipflop controlled by locally generated clock signal

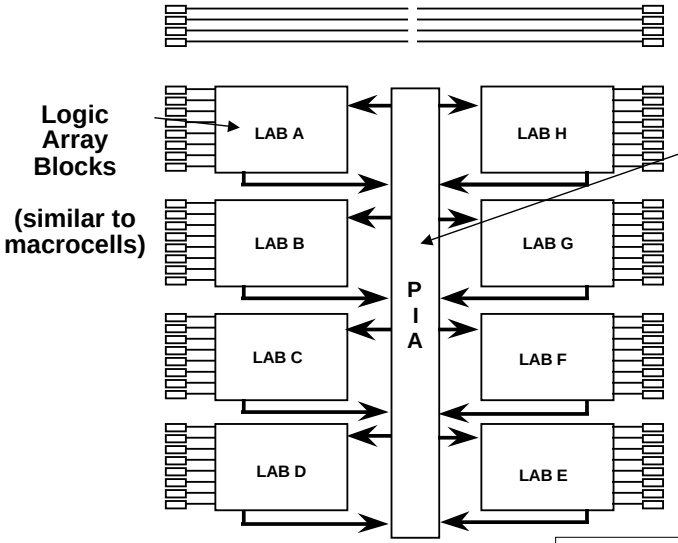
+ Seq Logic: could be D, T positive or negative edge triggered
+ product term to implement clear function

Implementation Strategies

Design with More Sophisticated PLDs

AND-OR structures are relatively limited
Cannot share signals/product terms among macrocells

Altera solution: Multiple Array Matrix (MAX)



Logic Array Blocks

(similar to macrocells)

Global Routing: Programmable Interconnect Array

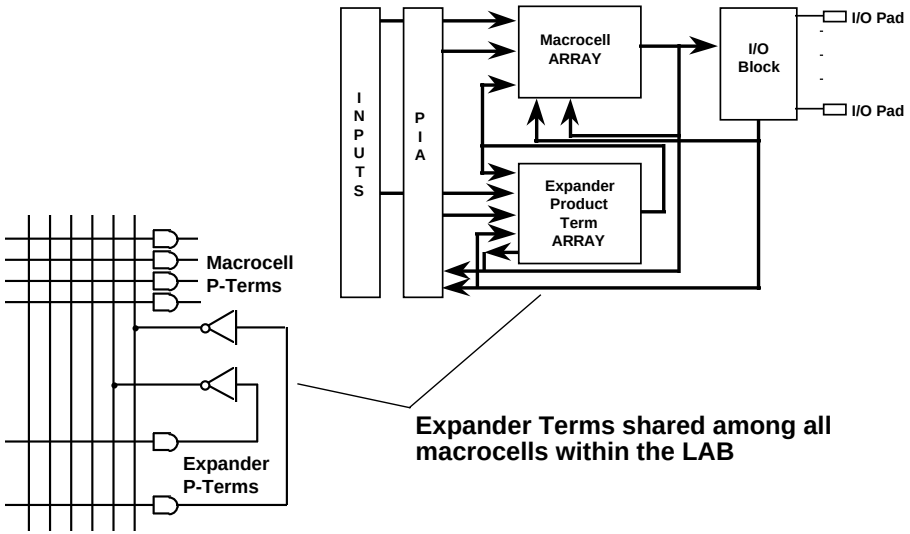
EPM5128:

8 Fixed Inputs
52 I/O Pins
8 LABs
16 Macrocells/LAB
32 Expanders/LAB

Implementation Strategies

Design with More Sophisticated PLDs

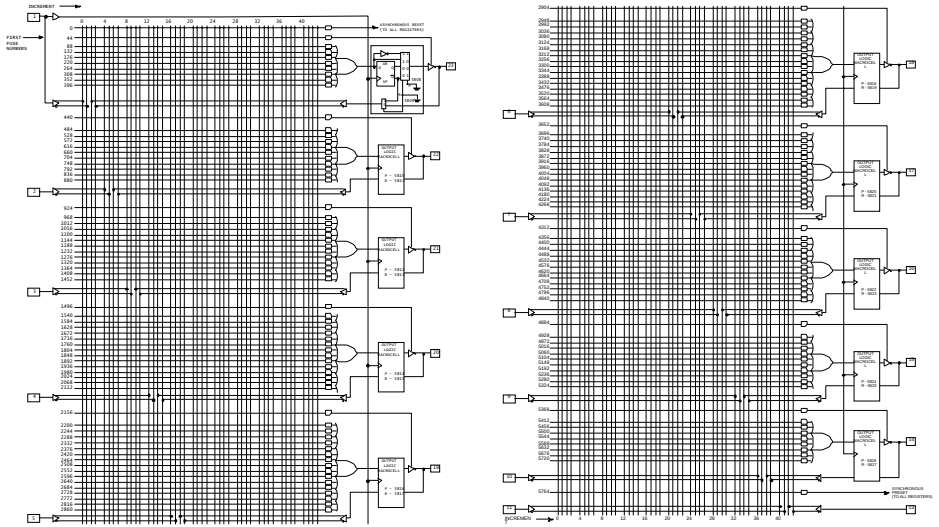
LAB Architecture



Expander Terms shared among all macrocells within the LAB

Implementation Strategies

P22V10 PAL



Supports large number of product terms per output
Latches and muxes associated with output pins

Implementation Strategies

Design with More Sophisticated PLDs

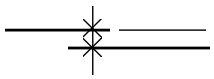
Actel Programmable Gate Arrays

Rows of programmable logic building blocks

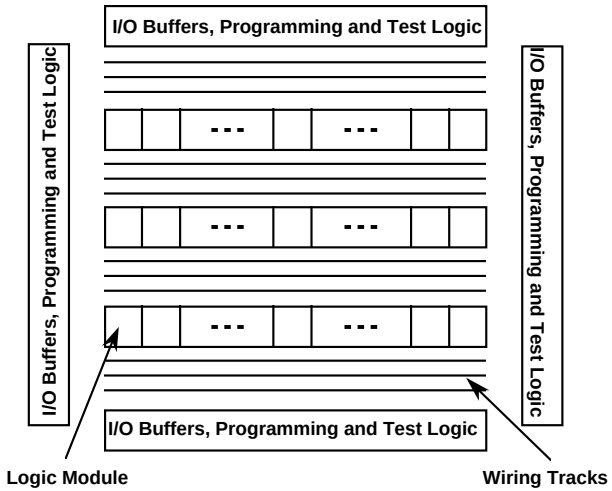
+

rows of interconnect

Anti-fuse Technology: Program Once



Use Anti-fuses to build up long wiring runs from short segments

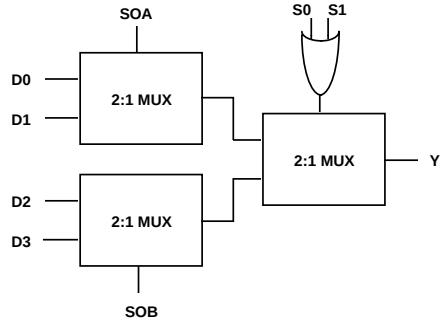


8 input, single output combinational logic blocks

FFs constructed from discrete cross coupled gates

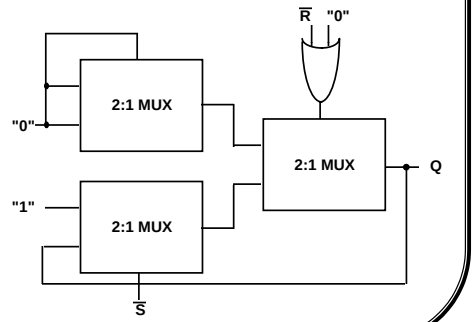
Implementation Strategies

Actel Logic Module



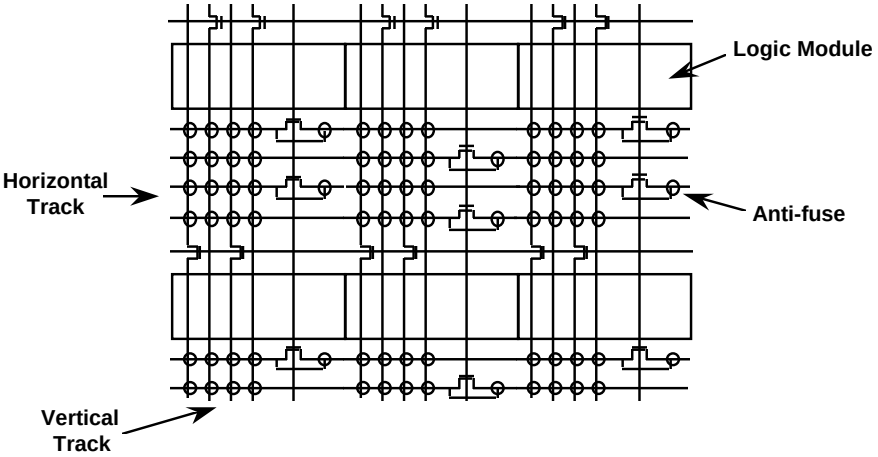
Basic Module is a Modified 4:1 Multiplexer

Example:
Implementation of S-R Latch $\overline{S}$ $\overline{R}$



Implementation Strategies

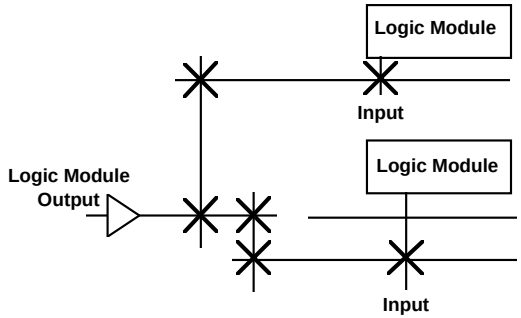
Actel Interconnect



Interconnection Fabric

Implementation Strategies

Actel Routing Example



Jogs cross an anti-fuse

minimize the # of jobs for speed critical circuits

2 - 3 hops for most interconnections

Implementation Strategies

Design with More Sophisticated PLDs

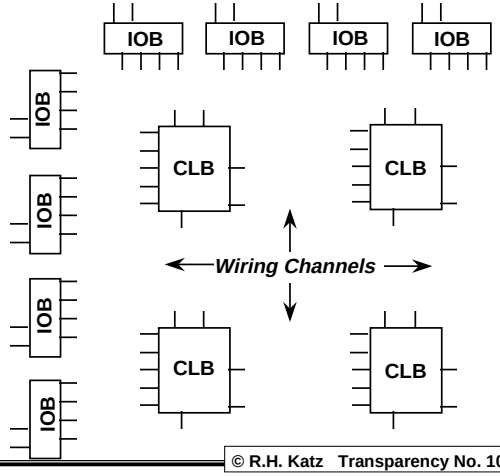
Xilinx Logic Cell Arrays

CMOS Static RAM Technology: programmable on the fly!

All personality elements connected into serial shift register

Shift in string of 1's and 0's on power up

General Chip Architecture:
- **Logic Blocks (CLBs)**
- **IO Blocks (IOBs)**
- **Wiring Channels**



Implementation Strategies

Design with More Sophisticated PLDs

Xilinx LCA Architecture

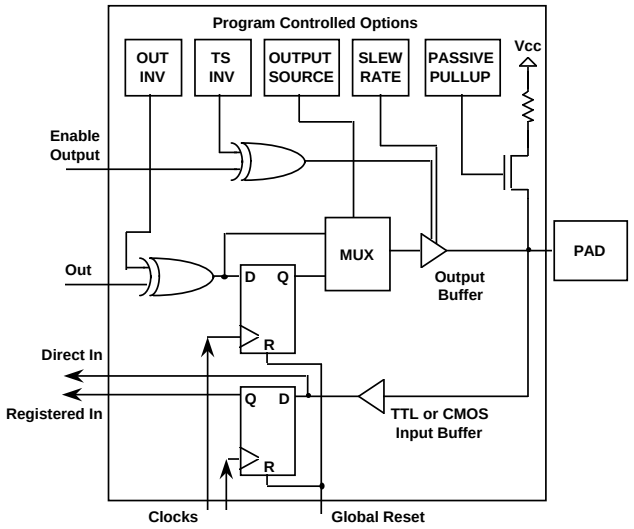
Inputs:
Tri-state enable
bit to output
input, output clocks

Outputs:
input bit

Internal FFs for
input & output paths

Fast/Slow outputs
5 ns vs. 30 ns rise

Pull-up used with
unused IOBs



Implementation Strategies

Design with More Sophisticated PLDs

Xilinx LCA Architecture

Configurable Logic Block: CLB

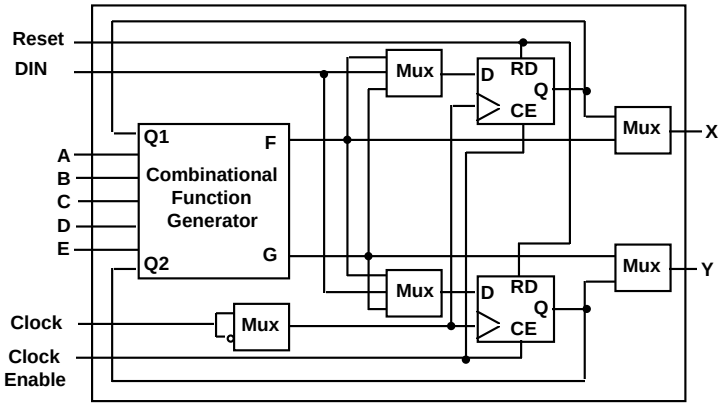
2 FFs

Any function of 5 Variables

Global Reset

Clock, Clock Enb

Independent DIN

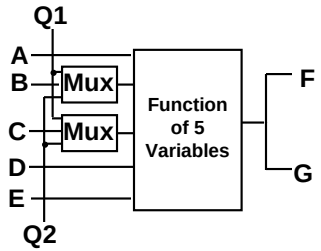


Implementation Strategies

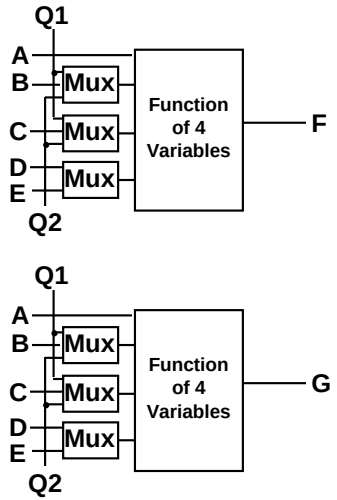
Design with More Sophisticated PLDs

Xilinx LCA Architecture

CLB Function Generator



Any function of 5 variables



Two Independent Functions of 4 variables each

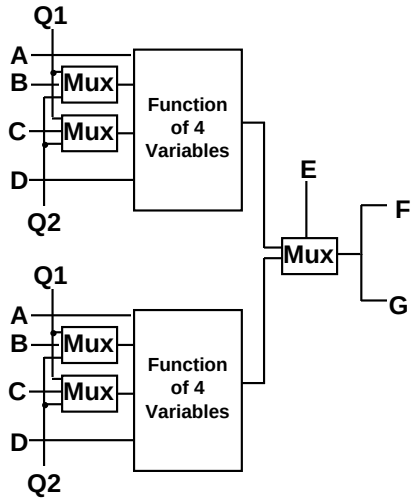
Implementation Strategies

Design with More Sophisticated PLDs

Xilinx LCA Architecture

CLB Function Generator

Certain Limited Functions of 6 Variables



Implementation Strategies

Designing with More Sophisticated PLDs

Xilinx Application Examples

5-Input Parity Generator

Implemented with 1 CLB:

$$F = \overline{A \text{ xor } B \text{ xor } C \text{ xor } D \text{ xor } E}$$

(this is a different parity generator than the one in Chapter 8!)

2-bit Comparator: A B = C D or A B > C D

Implemented with 1 CLB:

$$(GT)\ F = A\overline{C} + AB\overline{D} + B\overline{C}\,\overline{D}$$

$$(EQ)\ G = \overline{A}\,\overline{B}\,\overline{C}\,\overline{D} + \overline{A}B\overline{C}D + A\overline{B}C\overline{D} + ABCD$$

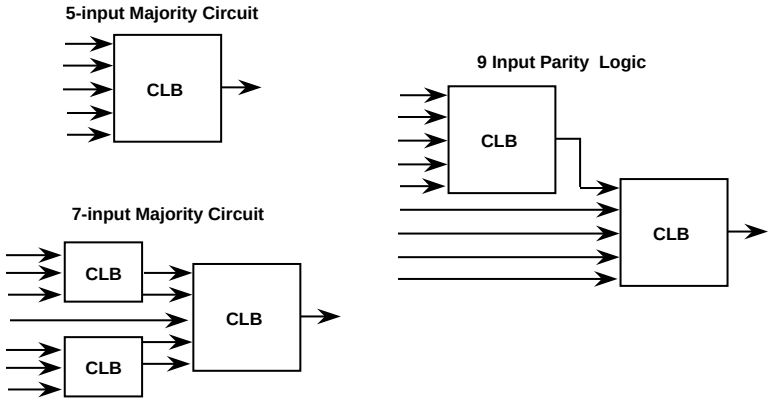
Implementation Strategies

Designing with More Sophisticated PLDs

Xilinx Application Examples

n-Input Majority Circuit

Assert 1 whenever n/2 or greater inputs are 1



n-input Parity Functions

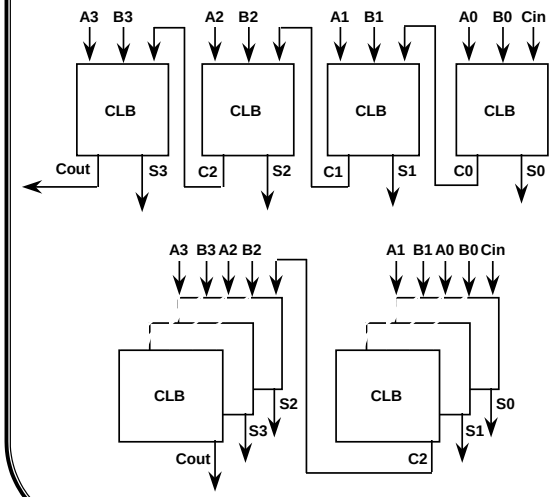
5 input = 1 CLB, 2 Levels of CLBs yield up to 25 inputs!

Implementation Strategies

Designing with More Sophisticated PLDs

Xilinx Application Examples

4-bit Binary Adder



Full Adder, 4 CLB delays to final carry out

2 x Two-bit Adders (3 CLBs each) yields 2 CLBs to final carry out

Implementation Strategies

Design with More Sophisticated PLDs

Xilinx LCA Architecture

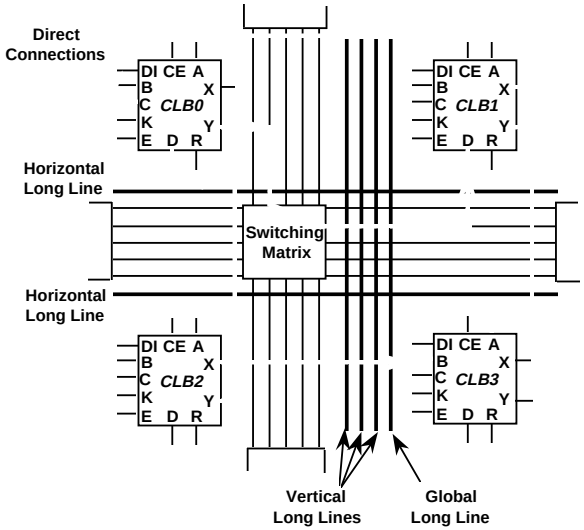
Interconnect

Direct Connections

Global Long Line

Horizontal/Vertical Long Lines

Switching Matrix Connections



Implementation Strategies

Design with More Sophisticated PLDs

Xilinx LCA Architecture

Implementing the BCD to Excess 3 FSM

$$Q2+ = \overline{Q2} \cdot Q0 + Q2 \cdot \overline{Q0}$$

$$Q1+ = \overline{X} \cdot \overline{Q2} \cdot \overline{Q1} \cdot Q0 + X \cdot \overline{Q2} \cdot Q0 + X \cdot Q2 \cdot \overline{Q0} + Q1 \cdot \overline{Q0}$$

$$Q0+ = \overline{Q0}$$

$$Z = Z \cdot Q1 + \overline{X} \cdot \overline{Q1}$$

No function more complex than 4 variables
4 FFs implies 2 CLBs

Synchronous Mealy Machine

Global Reset to be used

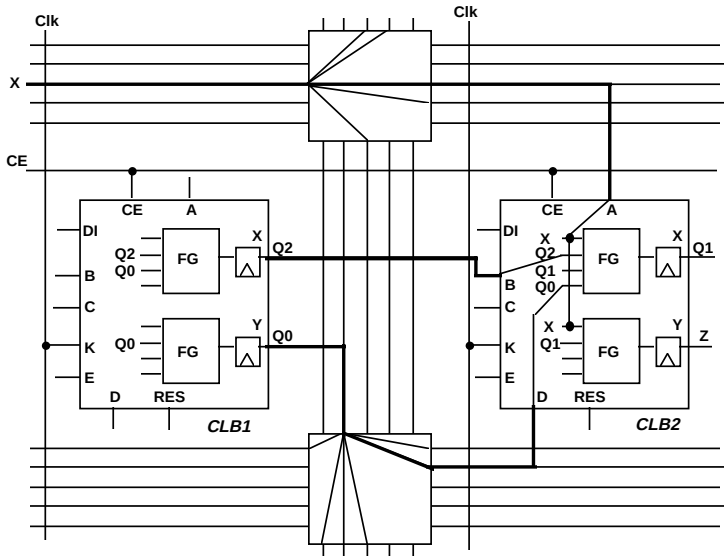
Place Q2+, Q0+ in once CLB
Q1, Z in second CLB
maximize use of direct & general purpose interconnections

Implementation Strategies

Design with More Sophisticated PLDs

Xilinx LCA Architecture

Implementing the BCD to Excess 3 FSM



Design Case Study

Traffic Light Controller

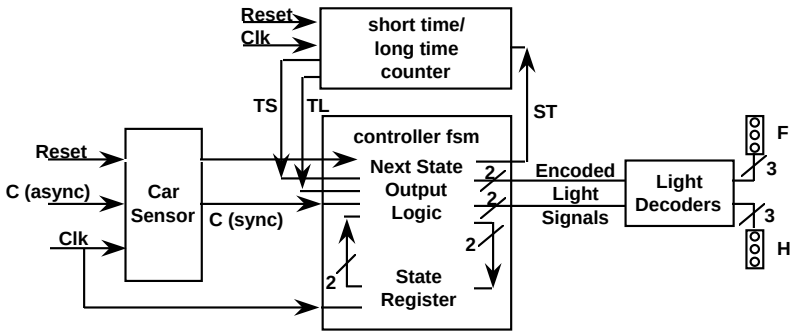
Decomposition into primitive subsystems

- Controller FSM
  - next state/output functions
  - state register
- Short time/long time interval counter
- Car Sensor
- Output Decoders and Traffic Lights

Design Case Study

Traffic Light Controller

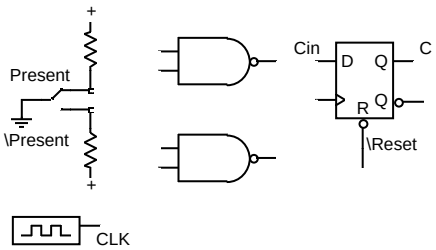
Block Diagram



Design Case Study

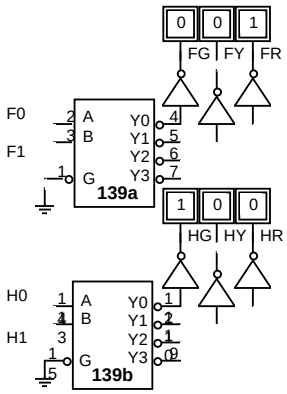
Traffic Light Controller

Subsystem Logic

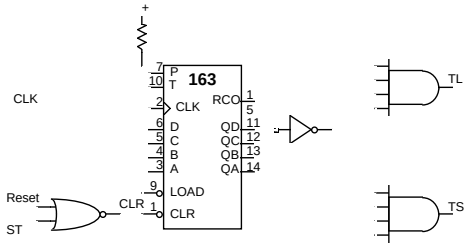


Car Detector

Light Decoders



Interval Timer



Design Case Study

Traffic Light Controller

Next State Logic

State Assignment: HG = 00, HY = 10, FG = 01, FY = 11

$P1 = C\ TL\ \overline{Q1} + TS\ Q1\ \overline{Q0} + \overline{C}\ \overline{Q1}\ Q0 + \overline{TS}\ Q1\ Q0$

$P0 = TS\ Q1\ \overline{Q0} + \overline{Q1}\ Q0 + \overline{TS}\ Q1\ Q0$

$ST = C\ TL\ \overline{Q1} + \overline{C}\ \overline{Q1}\ Q0 + TS\ Q1\ \overline{Q0} + TS\ Q1\ Q0$

$HL[1] = TS\ Q1\ Q0 + \overline{Q1}\ Q0 + \overline{TS}\ Q1\ Q0$

$HL[0] = \overline{TS}\ Q1\ \overline{Q0} + TS\ Q1\ \overline{Q0}$

$FL[1] = \overline{Q0}$

$FL[0] = TS\ Q1\ Q0 + \overline{TS}\ Q1\ Q0$

PAL/PLA Implementation:
- 5 inputs, 7 outputs, 8 product terms
- PAL 22V10 -- 11 inputs, 10 prog. IOs, 8 to 14 prod terms per OR

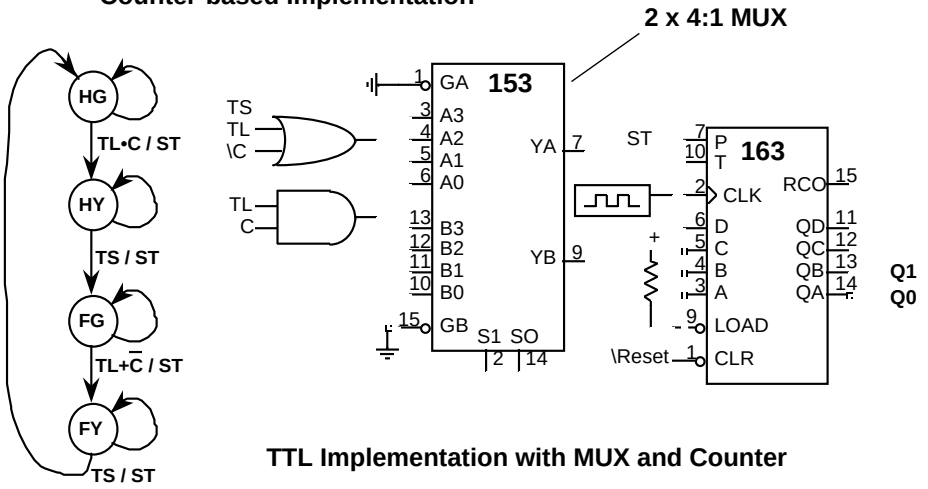
ROM Implementation:
- 32 word by 8-bit ROM (256 bits)
- Reset may double ROM size

Design Case Study

Traffic Light Controller

Next State Logic

Counter-based Implementation



TTL Implementation with MUX and Counter

Can we reduce package count by using an 8:1 MUX?

ST = Count

Design Case Study

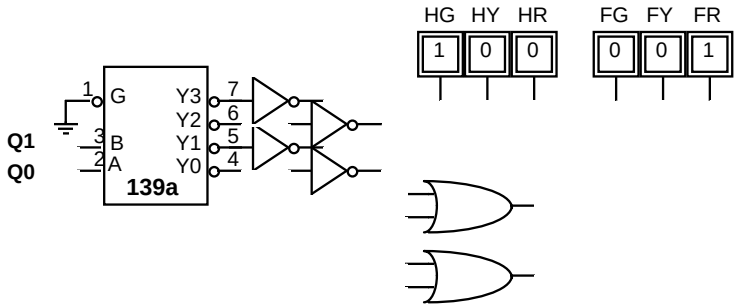
Traffic Light Controller

Next State Logic

Counter-based Implementation

Dispense with direct output functions for the traffic lights

Why not simply decode from the current state?



ST is a Synchronous Mealy Output

Light Controllers are Moore Outputs

Design Case Study

Traffic Light Controller

LCA-Based Implementation

Discrete Gate Method:

None of the functions exceed 5 variables

P1, ST are 5 variable (1 CLB each)

P0, HL1, HL0, FL0 are 3 variable (1/2 CLB each)

FL1 is 1 variable (1/2 CLB)

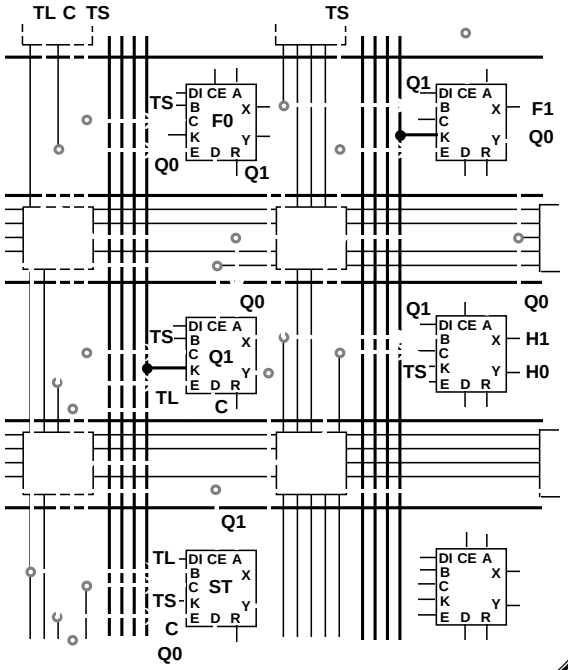
4 1/2 CLBs total!

Design Case Study

Traffic Light Controller

LCA-Based Implementation

Placement of functions selected to maximize the use of direct connections



Design Case Study

Traffic Light Controller

LCA-Based Implementation

Counter/Multiplexer Method:

4:1 MUX, 2 Bit Upcounter

MUX: six variables (4 data, 2 control)
but this is the kind of 6 variable function that can be implemented in 1 CLB!

2nd CLB to implement TL " C and TL + C'

But note that ST/Cnt is really a function of TL, C, TS, Q1, Q0
1 CLB to implement this function of 5 variables!

2 Bit Counter: 2 functions of 3 variables (2 bit state + count)
Also implemented in one CLB

Traffic light decoders: functions of 2 variables (Q1, Q0)
2 per CLB = 3 CLB for the six lights

Total count = 5 CLBs

Chapter Summary

" **Optimization and Implementation of FSM**

State Reduction Methods: Row Matching, Implication Chart

State Assignment Methods: Heuristics and Computer Tools

" **Implementation Issues**

Choice of Flipflops

Structured Logic Methods

ROM based

PLA/PAL based

Jump Counter Methods

Sophisticated PLDs: Altera, Actel, Xilinx