

# **CH-10: Sequential Logic** **Examples**

*Contemporary Logic Design*

YONSEI UNIVERSITY

Fall 2016

# Sequential Logic Examples

---

- Basic design approach: 4-step design process
- Hardware description languages and finite state machines
- Implementation examples and case studies
  - ◆ finite-string pattern recognizer
  - ◆ complex counter
  - ◆ traffic light controller
  - ◆ door combination lock
  - ◆ Serial Line Transmitter/Receiver

# General FSM Design Procedure

---

- (1) *Determine inputs and outputs*
- (2) *Determine possible states of machine*
  - ◆ state minimization
- (3) *Encode states and outputs into a binary code*
  - ◆ state assignment or state encoding
  - ◆ output encoding
  - ◆ possibly input encoding (if under our control)
- (4) *Realize logic to implement functions for states and outputs*
  - ◆ combinational logic implementation and optimization
  - ◆ choices in steps 2 and 3 can have large effect on resulting logic

# Finite String Pattern Recognizer (step 1)

---

- Finite string pattern recognizer
  - ◆ one input (X) and one output (Z)
  - ◆ 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: 0 0 1 0 1 0 1 0 0 1 0 ...

Z: 0 0 0 1 0 1 0 1 0 0 0 ...

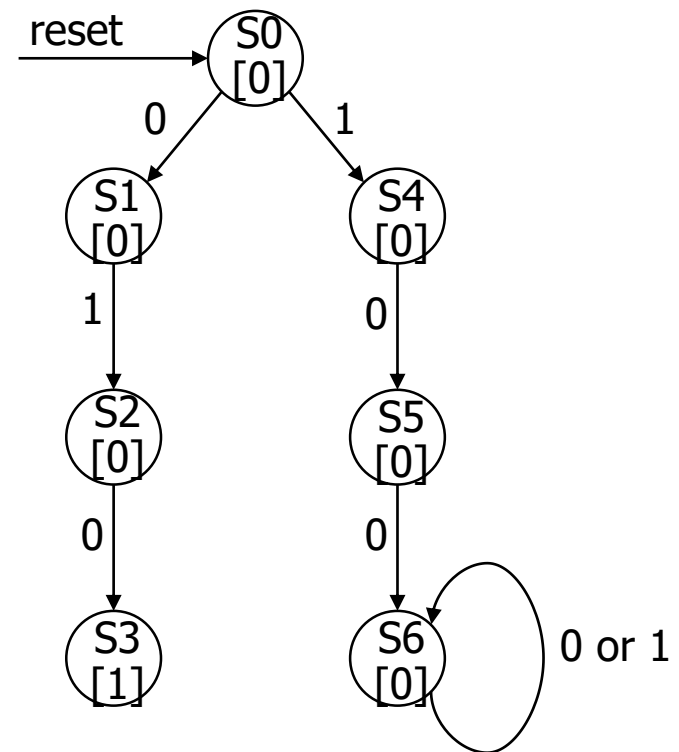
X: 1 1 0 1 1 0 1 0 0 1 0 ...

Z: 0 0 0 0 0 0 0 1 0 0 0 ...

# Finite String Pattern Recognizer (step 2)

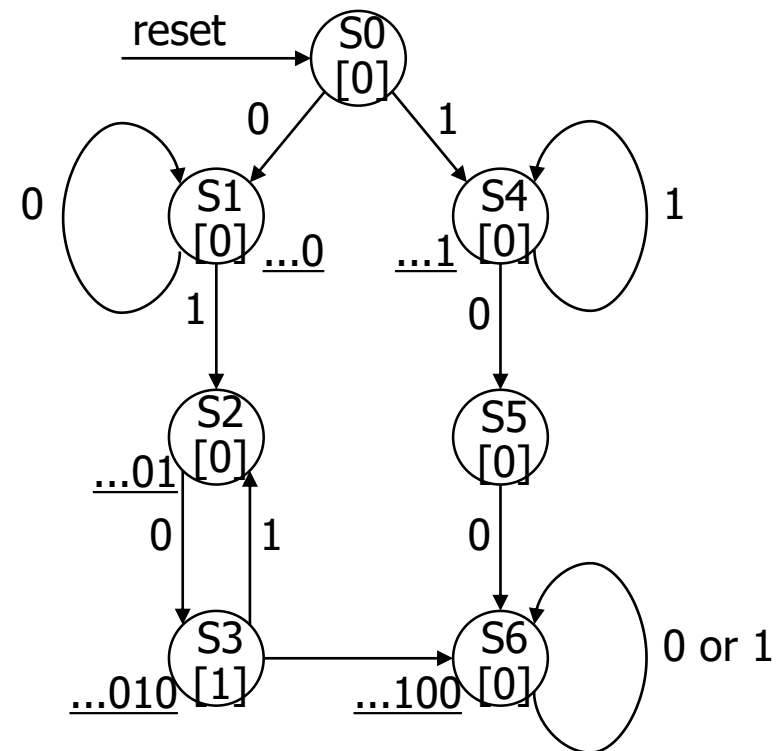
## ■ Step 2: *draw state diagram*

- ◆ for the strings that must be recognized, i.e., 010 and 100
- ◆ a Moore implementation



# Finite String Pattern Recognizer (step2)

- *Exit conditions from state S3:* have recognized ...010
  - ◆ if next input is 0 then have ...0100 = ...100 (state S6)
  - ◆ if next input is 1 then have ...0101 = ...01 (state S2)
- *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 String Pattern Recognizer (step 2)

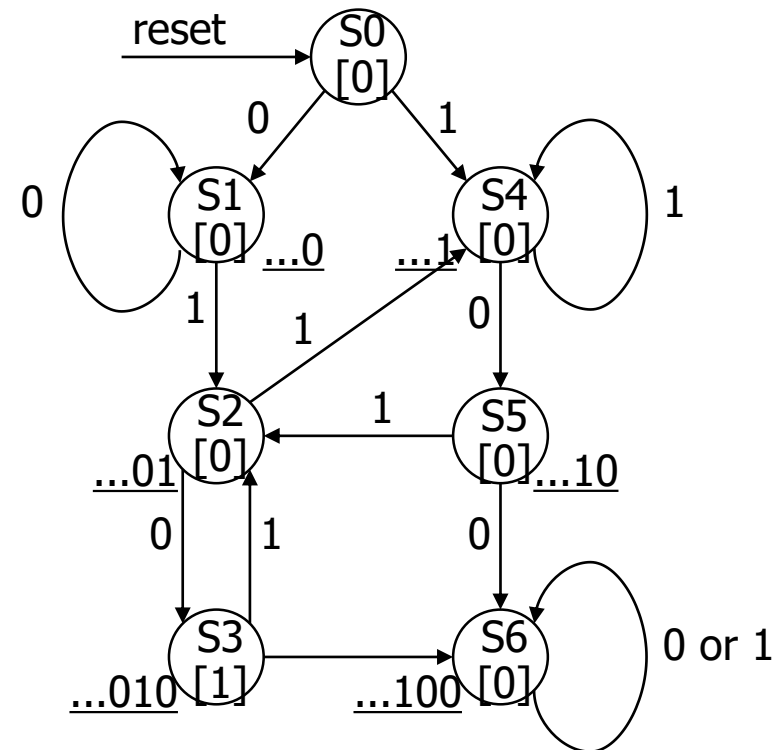
- S2 and S5 still have incomplete transitions

- ◆ 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

- *Reuse states as much as possible*

- ◆ look for same meaning
- ◆ state minimization leads to smaller number of bits to represent states

- Once all states have a complete set of transitions we have a final state diagram



# Finite String Pattern Recognizer (step 3)

- Verilog description including state assignment (or state encoding)

```
module string (clk, X, rst, Q0, Q1, Q2, Z);
input clk, X, rst;
output Q0, Q1, Q2, Z;

parameter S0 = [0,0,0]; //reset state
parameter S1 = [0,0,1]; //strings ending in ...0
parameter S2 = [0,1,0]; //strings ending in ...01
parameter S3 = [0,1,1]; //strings ending in ...010
parameter S4 = [1,0,0]; //strings ending in ...1
parameter S5 = [1,0,1]; //strings ending in ...10
parameter S6 = [1,1,0]; //strings ending in ...100

reg state[0:2];

assign Q0 = state[0];
assign Q1 = state[1];
assign Q2 = state[2];
assign Z = (state == S3);
```

```
always @(posedge clk) begin
    if (rst) state = S0;
    else
        case (state)
            S0: if (X) state = S4 else state = S1;
            S1: if (X) state = S2 else state = S1;
            S2: if (X) state = S4 else state = S3;
            S3: if (X) state = S2 else state = S6;
            S4: if (X) state = S4 else state = S5;
            S5: if (X) state = S2 else state = S6;
            S6: state = S6;
            default: begin
                $display ("invalid state reached");
                state = 3'bxxx;
            end
        endcase
    end
end

endmodule
```



# Finite String Pattern Recognizer

---

- Review of process
  - ◆ understanding problem
    - write down sample inputs and outputs to understand specification
  - ◆ derive a state diagram
    - write down sequences of states and transitions for sequences to be recognized
  - ◆ minimize number of states
    - add missing transitions; reuse states as much as possible
  - ◆ state assignment or encoding
    - encode states with unique patterns
  - ◆ simulate realization
    - verify I/O behavior of your state diagram to ensure it matches specification

# Finite String Pattern Recognizer (Step 4)

## ■ Implementation

- ◆ Encoded state table and K-maps for the string recognizer

$CS_0$	$CS_1$	$CS_2$	$X$	$NS_0$	$NS_1$	$NS_2$
0	0	0	0	0	0	1
0	0	0	1	1	0	0
0	0	1	0	0	0	1
0	0	1	1	0	1	0
0	1	0	0	0	1	1
0	1	0	1	1	0	0
0	1	1	0	1	1	0
0	1	1	1	0	1	0
1	0	0	0	1	0	1
1	0	0	1	1	0	0
1	0	1	0	1	1	0
1	0	1	1	0	1	0
1	1	0	0	1	1	0
1	1	0	1	1	1	0
1	1	1	0	X	X	X
1	1	1	1	X	X	X

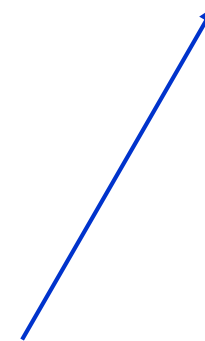


$$NS_0 = CS_0 Q'_2 + CS'_2 X + CS_1 CS_2 X' + CS_0 X'$$

$$NS_1 = CS_1 X' + CS_0 CS_2 + CS_0 CS_1 + CS_2 X$$

$$NS_2 = CS'_0 CS'_2 X' + CS'_1 CS'_2 X' + CS'_0 CS'_1 X'$$

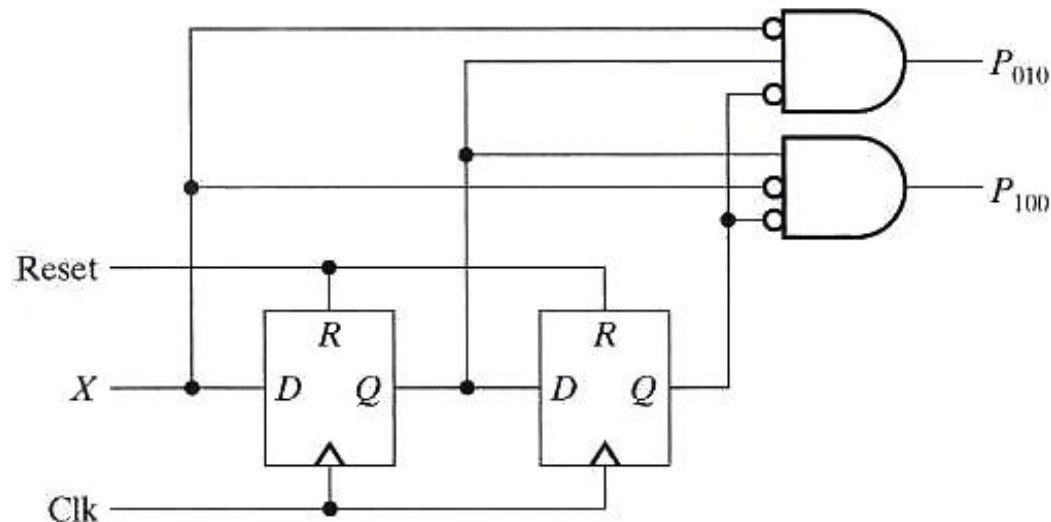
$$Z = CS'_0 CS_1 CS_2$$



These are fairly complex equations for a small FSM

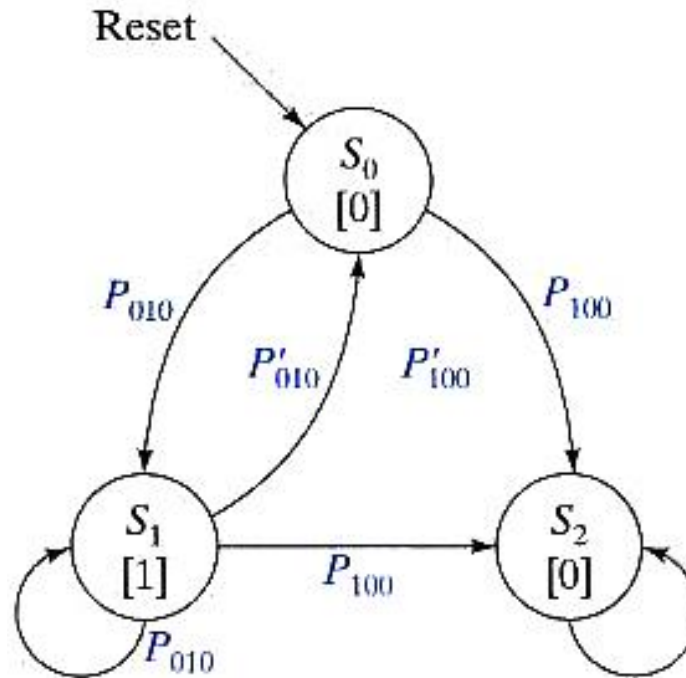
# Finite String Pattern Recognizer (Step 4)

- Better choice
  - ◆ Look at the last three values of the input: **A *shift register*** is perfect
  - ◆ Only need a 2-bit shift register to hold the two previous inputs
- Pattern recognition logic for the new string recognizer
  - ◆ add two 3-input gates that will check for the two patterns '010' and '100'



# Finite String Pattern Recognizer (Step 4)

- FSM for the new string recognizer



# Finite String Pattern Recognizer (Step 4)

- Encoded state table and K-maps for the new string recognizer

$CS_1$	$CS_2$	$P_{010}$	$P_{100}$	$NS_0$	$NS_1$
0	0	0	0	0	0
0	0	0	1	1	0
0	0	1	0	0	1
0	0	1	1	X	X
0	1	0	0	0	0
0	1	0	1	1	0
0	1	1	0	0	1
0	1	1	1	X	X
1	0	0	0	1	0
1	0	0	1	1	0
1	0	1	0	1	0
1	0	1	1	X	X
1	1	0	0	X	X
1	1	0	1	X	X
1	1	1	0	X	X
1	1	1	1	X	X



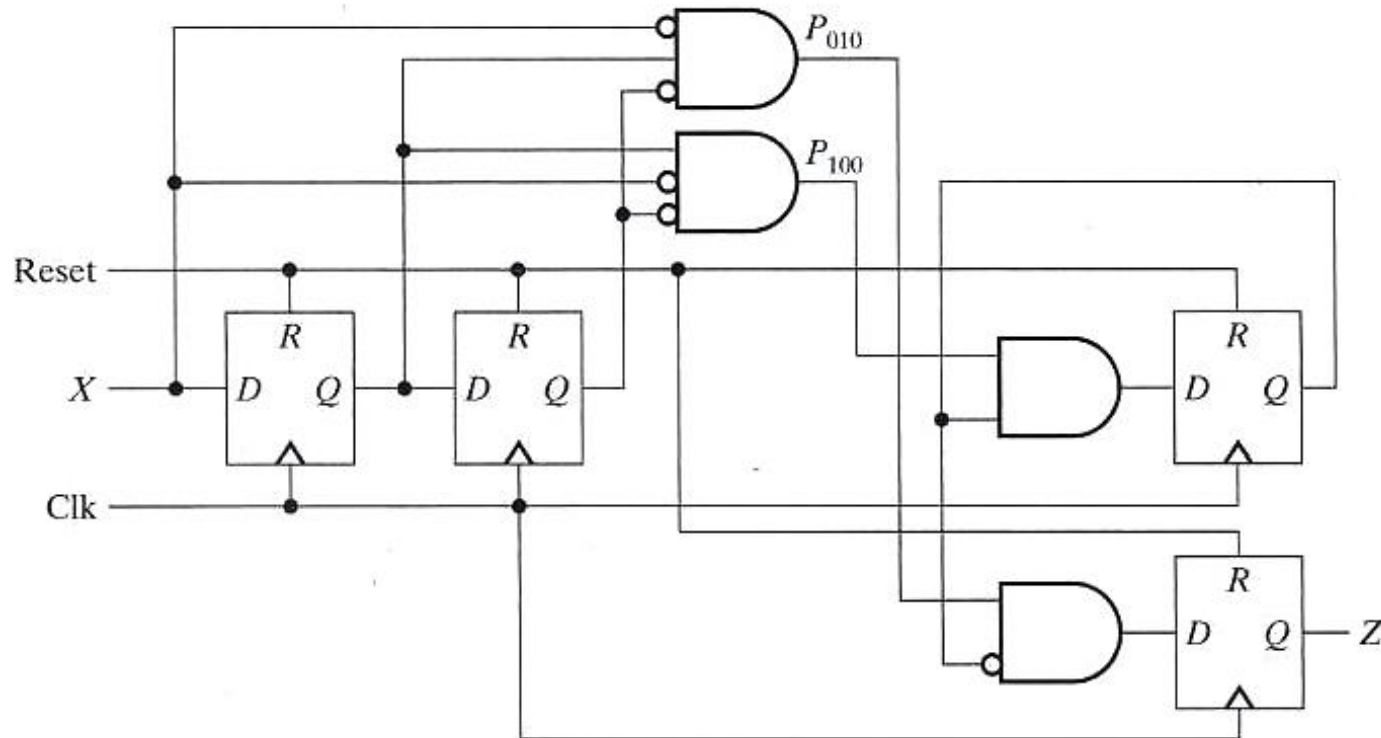
$$NS_0 = CS_0 + P_{100}$$

$$NS_1 = CS'_0 P_{010} + CS_1 P_{010} = CS'_0 P_{010}$$

$$Z = CS_1$$

# Finite String Pattern Recognizer (Step 4)

- Complete circuit for the new string recognizer



# Complex Counter

- A synchronous 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

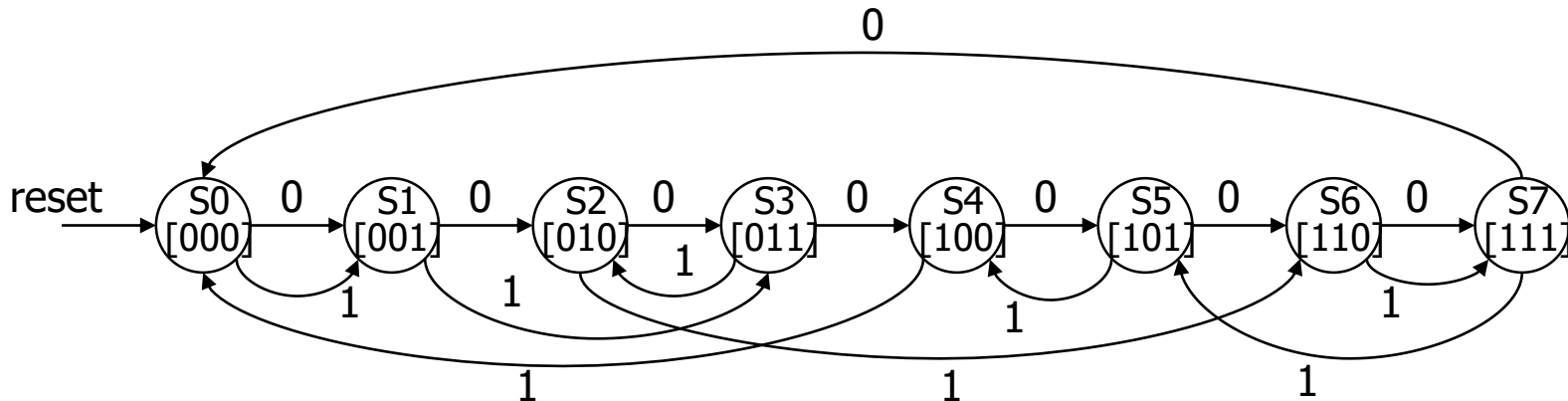
Gray: 000, 001, 011, 010, 110, 111, 101, 100

- Valid I/O behavior (partial)

Mode Input M	Current State	Next State
0	000	001
0	001	010
1	010	110
1	110	111
1	111	101
0	101	110
0	110	111

# Complex Counter (State Diagram)

- Deriving state diagram
  - ◆ one state for each output combination
  - ◆ add appropriate arcs for the mode control





# Complex Counter (State Encoding)

- Verilog description including state encoding

```
module string (clk, M, rst, Z0, Z1, Z2);  
  input clk, X, rst;  
  output Z0, Z1, Z2;
```

```
  parameter S0 = [0,0,0];  
  parameter S1 = [0,0,1];  
  parameter S2 = [0,1,0];  
  parameter S3 = [0,1,1];  
  parameter S4 = [1,0,0];  
  parameter S5 = [1,0,1];  
  parameter S6 = [1,1,0];  
  parameter S7 = [1,1,1];
```

```
  reg state[0:2];
```

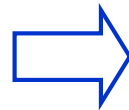
```
  assign Z0 = state[0];  
  assign Z1 = state[1];  
  assign Z2 = state[2];
```

```
  always @(posedge clk) begin  
    if rst state = S0;  
    else  
      case (state)  
        S0: state = S1;  
        S1: if (M) state = S3 else state = S2;  
        S2: if (M) state = S6 else state = S3;  
        S3: if (M) state = S2 else state = S4;  
        S4: if (M) state = S0 else state = S5;  
        S5: if (M) state = S4 else state = S6;  
        S6: if (M) state = S7 else state = S7;  
        S7: if (M) state = S5 else state = S0;  
      endcase  
    end  
  end  
endmodule
```

# Complex Counter (Implementation)

- Truth Table for computing the next Gray-code value and K-maps

$Z_2$	$Z_1$	$Z_0$	$D_2$	$D_1$	$D_0$
0	0	0	0	0	1
0	0	1	0	1	1
0	1	0	1	1	0
0	1	1	0	1	0
1	0	0	0	0	0
1	0	1	1	0	0
1	1	0	1	1	1
1	1	1	1	0	1



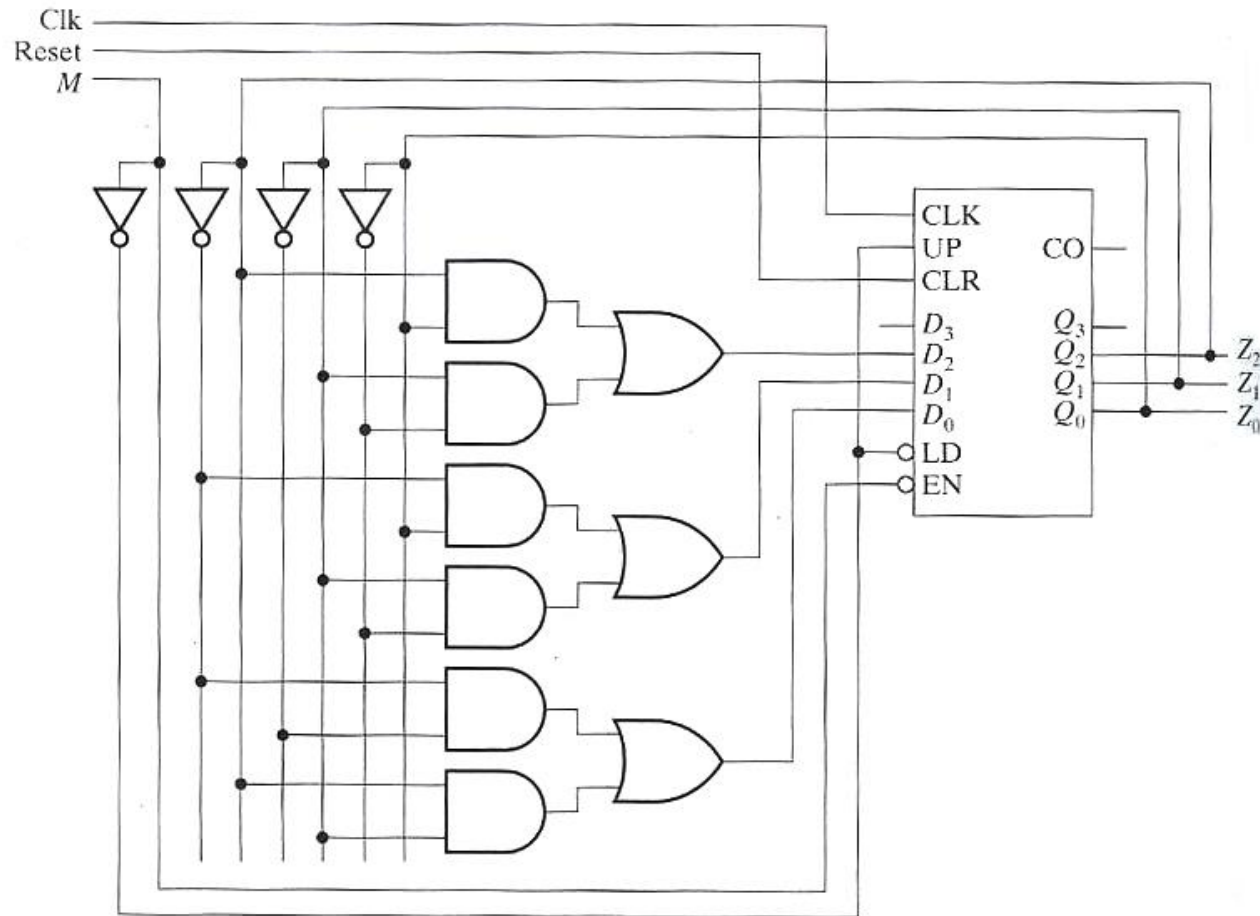
$$D_2 = Z_2 Z_0 + Z_1 Z'_0$$

$$D_1 = Z'_2 Z_0 + Z_1 Z'_0$$

$$D_0 = Z'_2 Z'_1 + Z_2 Z_1$$

# Complex Counter (Implementation)

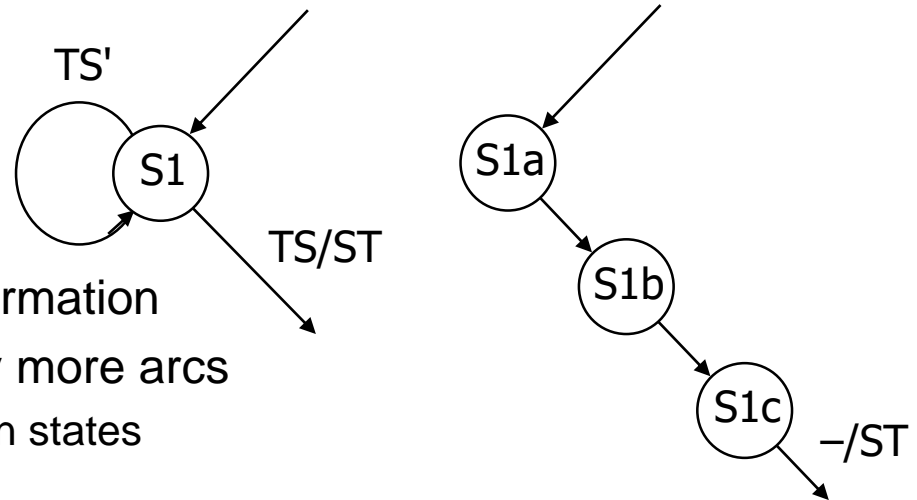
- Complete circuit for complex counter using a binary counter component



# Traffic LC as 2 Communicating FSMs

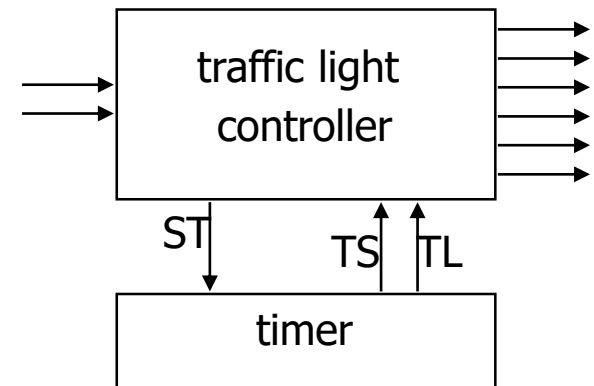
## ■ Without separate timer

- ◆ S0 would require 7 states
- ◆ S1 would require 3 states
- ◆ S2 would require 7 states
- ◆ S3 would require 3 states
- ◆ S1 and S3 have simple transformation
- ◆ S0 and S2 would require many more arcs
  - C could change in any of seven states



## ■ By factoring out timer

- ◆ greatly reduce number of states
  - 4 instead of 20
- ◆ counter only requires seven or eight states
  - 12 total instead of 20



# Traffic Light Controller FSM

## ■ Specification of inputs, outputs, and state elements

```
module FSM(HR, HY, HG, FR, FY, FG, ST, TS, TL, C, reset, Clk);  
    output    HR;  
    output    HY;  
    output    HG;  
    output    FR;  
    output    FY;  
    output    FG;  
    output    ST;  
    input     TS;  
    input     TL;  
    input     C;  
    input     reset;  
    input     Clk;  
  
    reg [6:1] state;  
    reg      ST;  
  
    parameter highwaygreen   = 6'b001100;  
    parameter highwayyellow  = 6'b010100;  
    parameter farmroadgreen  = 6'b100001;  
    parameter farmroadyellow = 6'b100010;  
  
    assign HR = state[6];  
    assign HY = state[5];  
    assign HG = state[4];  
    assign FR = state[3];  
    assign FY = state[2];  
    assign FG = state[1];
```

The diagram shows three arrows originating from the 'state' variable definition 'reg [6:1] state;'. One arrow points to the 'assign' statements for HR, HY, HG, FR, FY, and FG. Another arrow points to the 'parameter' definitions for highwaygreen, highwayyellow, farmroadgreen, and farmroadyellow. A third arrow points to the 'reg ST;' line.

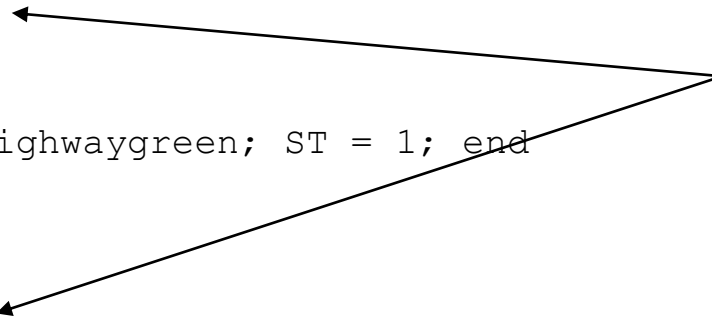
specify state bits and codes  
for each state as well as  
connections to outputs

# Traffic Light Controller FSM

```
initial begin state = highwaygreen; ST = 0; end

always @(posedge Clk)
begin
    if (reset)
        begin state = highwaygreen; ST = 1; end
    else
        begin
            ST = 0;
            case (state)
                highwaygreen:
                    if (TL & C) begin state = highwayyellow; ST = 1; end
                highwayyellow:
                    if (TS) begin state = farmroadgreen; ST = 1; end
                farmroadgreen:
                    if (TL | !C) begin state = farmroadyellow; ST = 1; end
                farmroadyellow:
                    if (TS) begin state = highwaygreen; ST = 1; end
            endcase
        end
    end
end
endmodule
```

case statement  
triggerred by  
clock edge



# Timer for Traffic Light Controller

---

## ■ Another FSM

```
module Timer(TS, TL, ST, Clk);
    output TS;
    output TL;
    input    ST;
    input    Clk;
    integer  value;

    assign TS = (value >= 4); // 5 cycles after reset
    assign TL = (value >= 14); // 15 cycles after reset

    always @(posedge ST) value = 0; // async reset

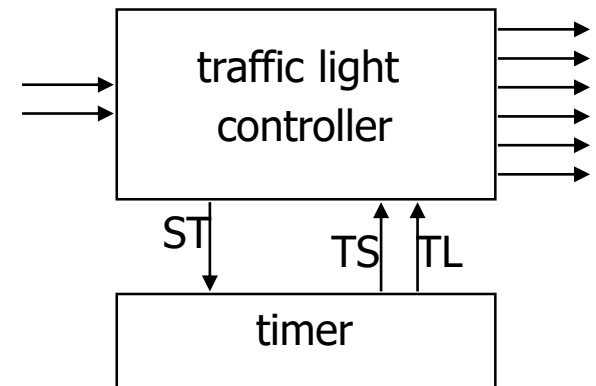
    always @(posedge Clk) value = value + 1;

endmodule
```

# Complete Traffic Light Controller

- Tying it all together (FSM + timer)
  - ◆ structural Verilog (same as a schematic drawing)

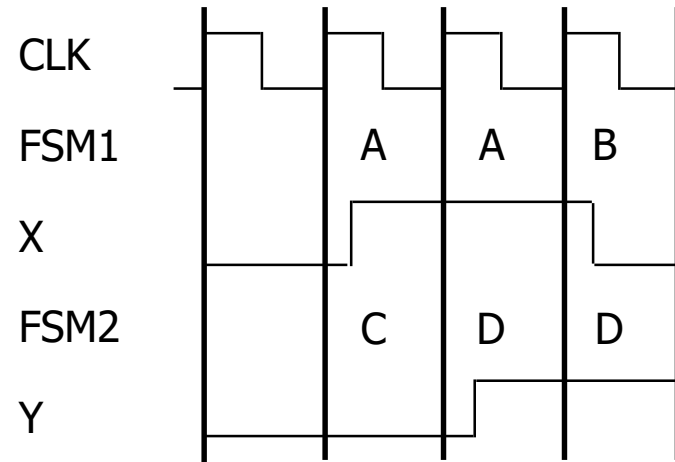
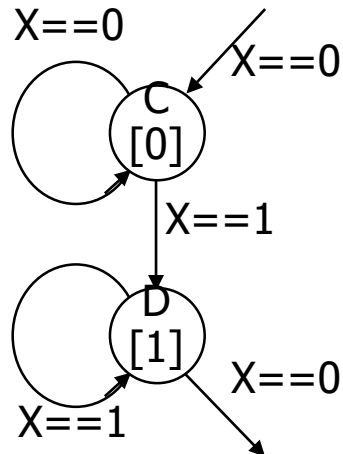
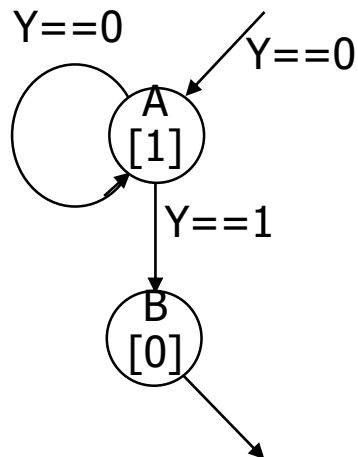
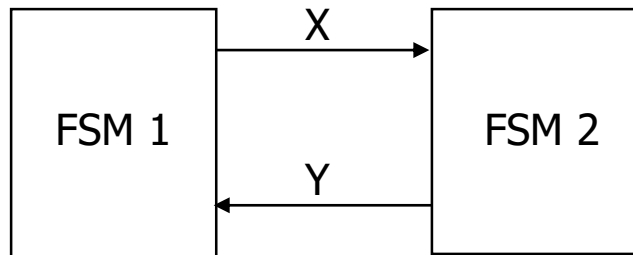
```
module main(HR, HY, HG, FR, FY, FG, reset, C, Clk);  
    output HR, HY, HG, FR, FY, FG;  
    input  reset, C, Clk;  
  
    Timer part1(TS, TL, ST, Clk);  
    FSM   part2(HR, HY, HG, FR, FY, FG, ST, TS, TL, C, reset, Clk);  
endmodule
```





# Communicating Finite State Machines

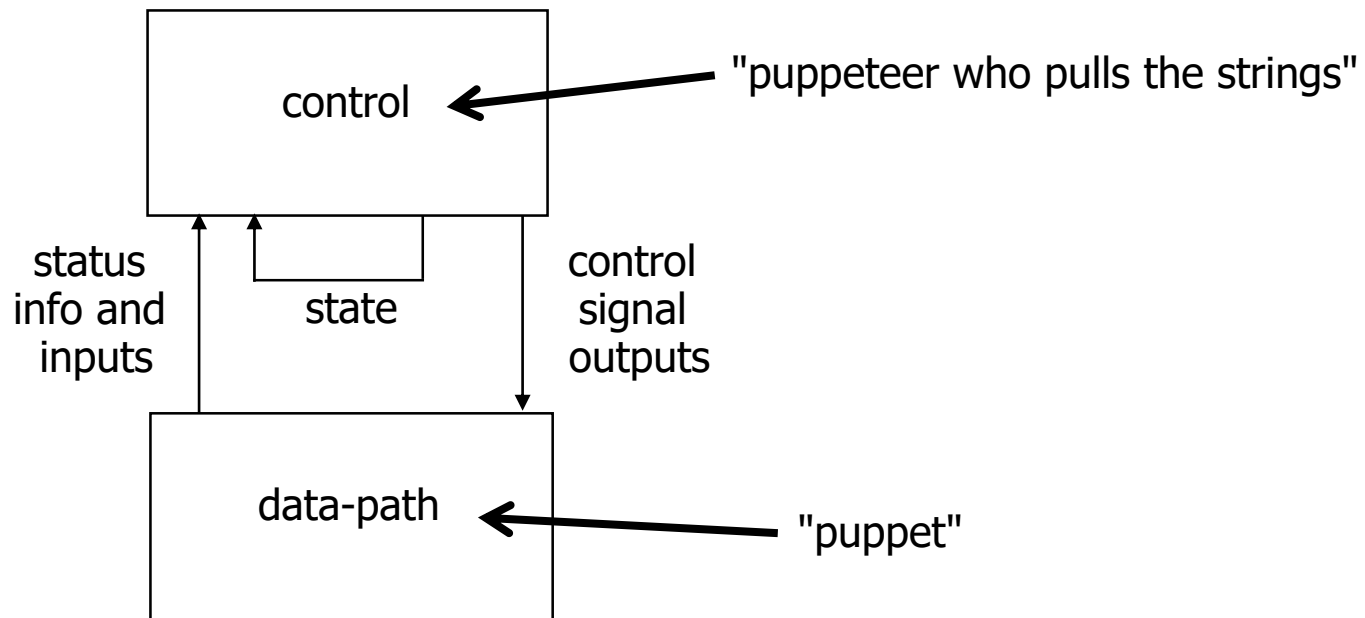
- One machine's output is another machine's input



machines advance in lock step  
initial inputs/outputs:  $X = 0$ ,  $Y = 0$

# Data-path and Control

- Digital hardware systems = data-path + control
  - ◆ datapath: registers, counters, combinational functional units (e.g., ALU), communication (e.g., busses)
  - ◆ control: FSM generating sequences of control signals that instructs datapath what to do next



# Digital Combinational Lock

---

- Door combination lock:
  - ◆ punch in 3 values in sequence and the door opens; if there is an error the lock must be reset; once the door opens the lock must be reset
  - ◆ inputs: sequence of input values, reset
  - ◆ outputs: door open/close
  - ◆ memory: must remember combination or always have it available
  - ◆ open questions: how do you set the internal combination?
    - stored in registers (how loaded?)
    - hardwired via switches set by user

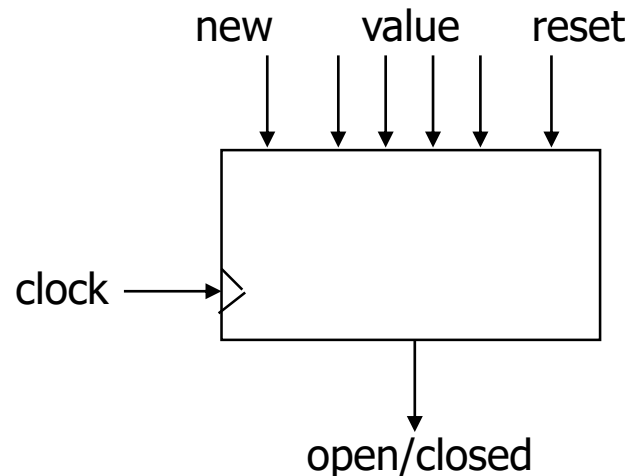
# Implementation in Software

---

```
integer combination_lock ( ) {  
    integer v1, v2, v3;  
    integer error = 0;  
    static integer c[3] = 3, 4, 2;  
  
    while (!new_value( ));  
    v1 = read_value( );  
    if (v1 != c[1]) then error = 1;  
  
    while (!new_value( ));  
    v2 = read_value( );  
    if (v2 != c[2]) then error = 1;  
  
    while (!new_value( ));  
    v3 = read_value( );  
    if (v2 != c[3]) then error = 1;  
  
    if (error == 1) then return(0); else return (1);  
}
```

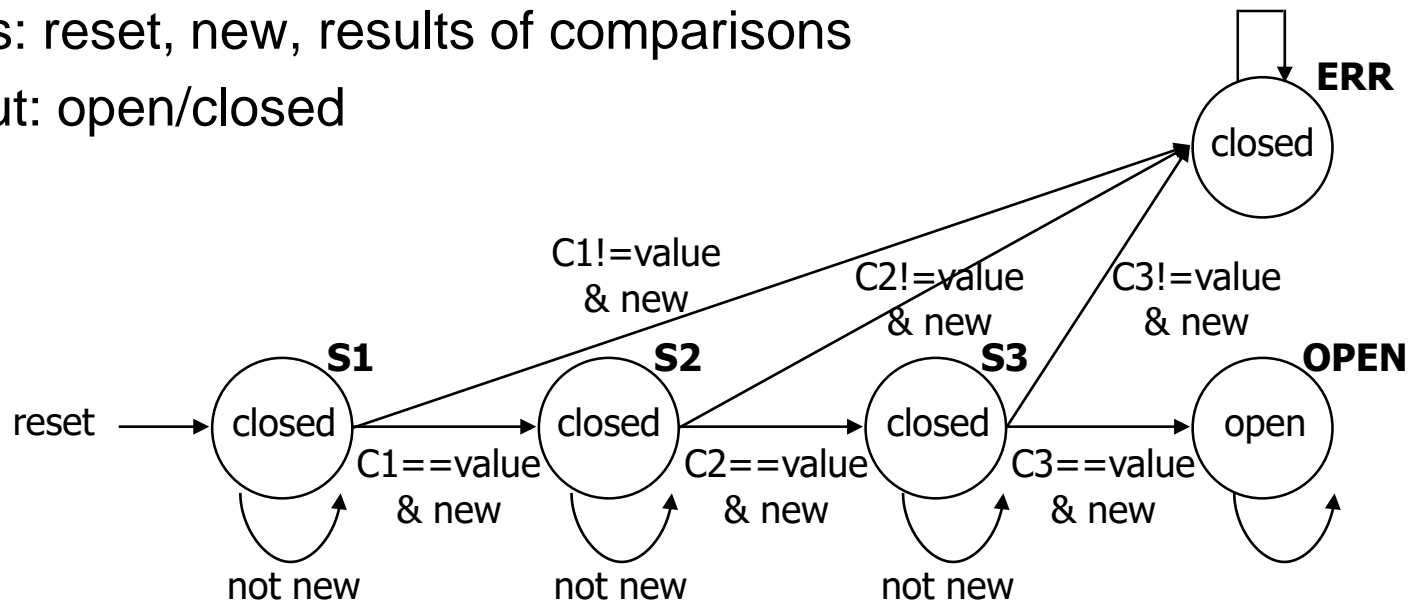
# Determining Details of Specification

- How many bits per input value?
- How many values in sequence?
- How do we know a new input value is entered?
- What are the states and state transitions of the system?



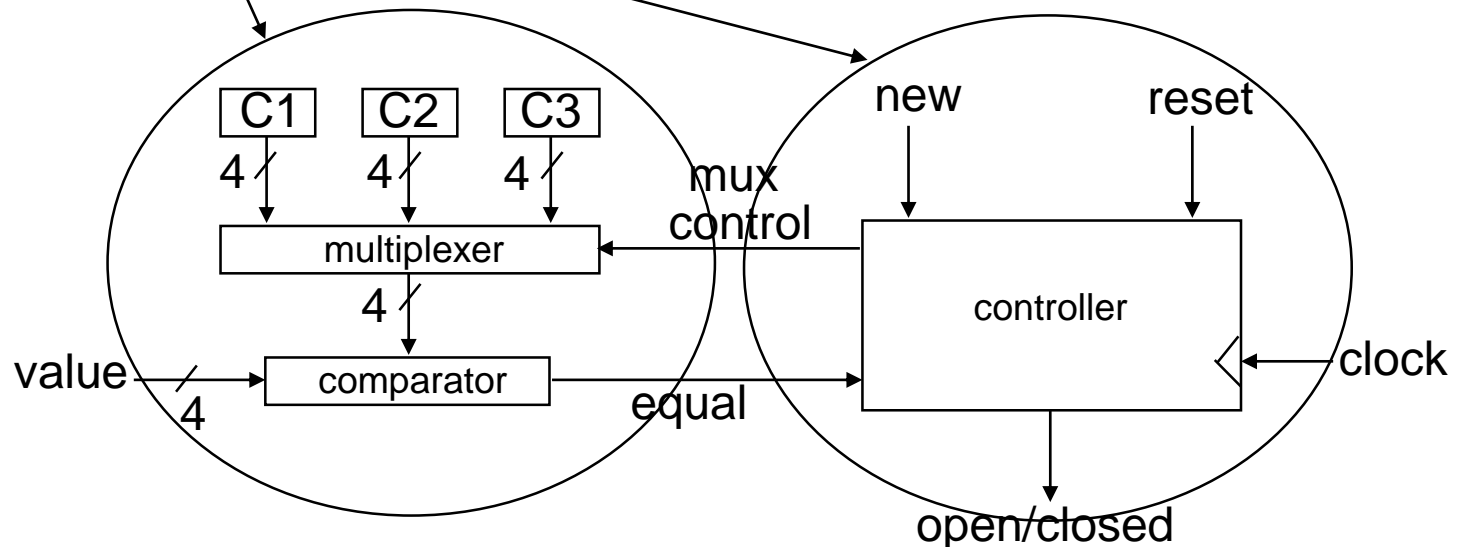
# Digital Combination Lock State Diagram

- States: 5 states
  - ◆ represent point in execution of machine
  - ◆ each state has outputs
- Transitions: 6 from state to state, 5 self transitions, 1 global
  - ◆ changes of state occur when clock says its ok
  - ◆ based on value of inputs
- Inputs: reset, new, results of comparisons
- Output: open/closed



# Data-path and Control Structure

- Data-path
  - ◆ storage registers for combination values
  - ◆ multiplexer
  - ◆ comparator
- Control
  - ◆ finite-state machine controller
  - ◆ control for data-path (which value to compare)



# State Table for Combination Lock

- Finite-state machine
  - ◆ refine state diagram to take internal structure into account
  - ◆ state table ready for encoding

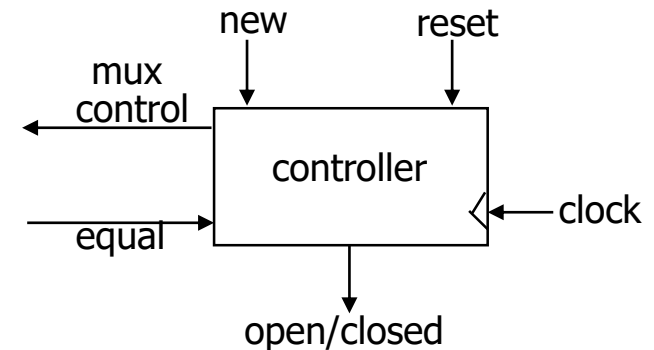
reset	new	equal	state	next state	mux	open/closed
1	–	–	–	S1	C1	closed
0	0	–	S1	S1	C1	closed
0	1	0	S1	ERR	–	closed
0	1	1	S1	S2	C2	closed
...						
0	1	1	S3	OPEN	–	open
...						



# Encodings for Combination Lock

## ■ Encode state table

- ◆ state can be: S1, S2, S3, OPEN, or ERR
  - needs at least 3 bits to encode: 000, 001, 010, 011, 100
  - and as many as 5: 00001, 00010, 00100, 01000, 10000
  - choose 4 bits: 0001, 0010, 0100, 1000, 0000
- ◆ output mux can be: C1, C2, or C3
  - needs 2 to 3 bits to encode
  - choose 3 bits: 001, 010, 100
- ◆ output open/closed can be: open or closed
  - needs 1 or 2 bits to encode
  - choose 1 bit: 1, 0

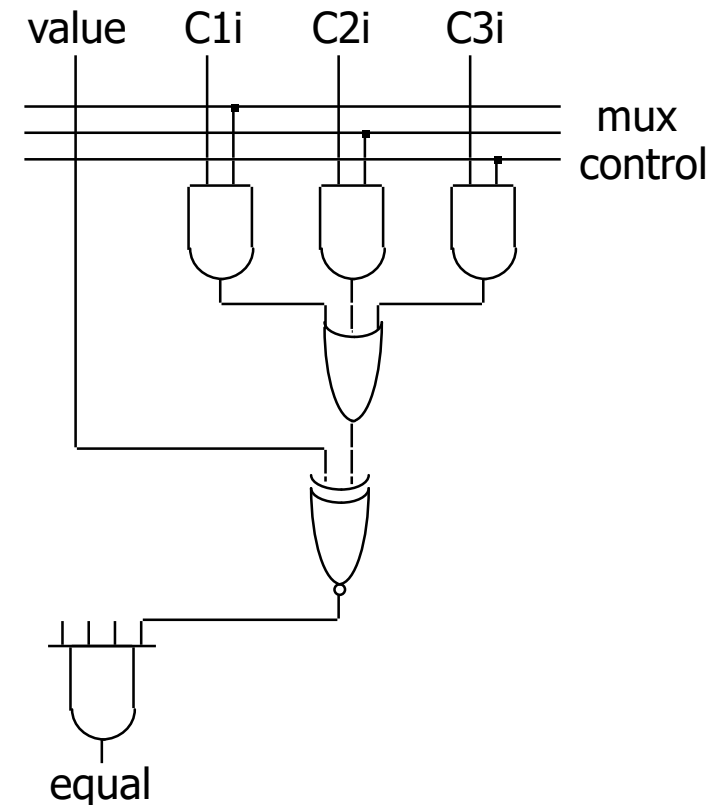
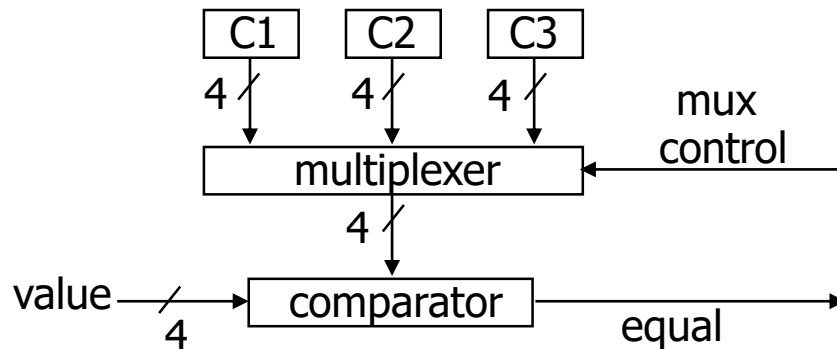


reset	new	equal	state	next state	mux	open/closed
1	—	—	—	0001	001	0
0	0	—	0001	0001	001	0
0	1	0	0001	0000	—	0
0	1	1	0001	0010	010	0
...						
0	1	1	0100	1000	—	1
...						

mux is identical to last 3 bits of state  
 open/closed is identical to first bit of state  
 therefore, we do not even need to implement  
 FFs to hold state, just use outputs

# Data-path Implementation: Comb-Lock

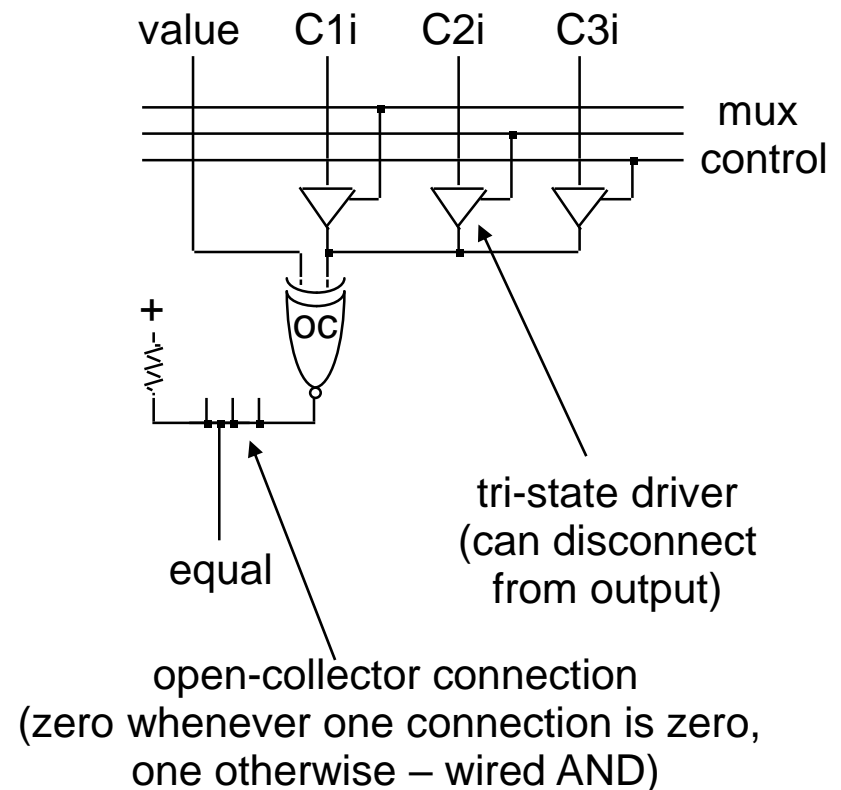
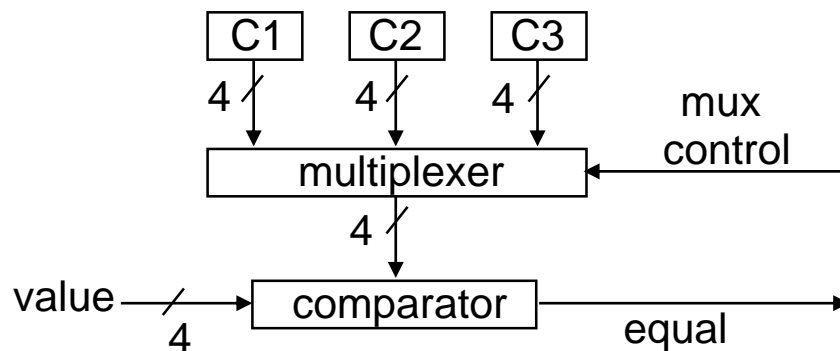
- Multiplexer
  - ◆ easy to implement as combinational logic when few inputs
  - ◆ logic can easily get too big for most PLDs



# Data-path Implementation

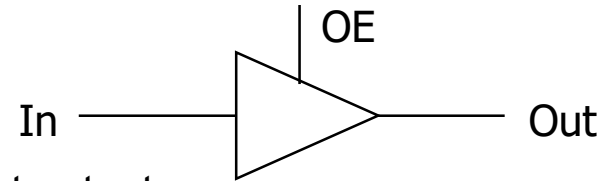
## ■ Tri-state logic

- ◆ utilize a third output state: “no connection” or “float”
- ◆ connect outputs together as long as only one is “enabled”
- ◆ open-collector gates can only output 0, not 1
  - can be used to implement logical AND with only wires

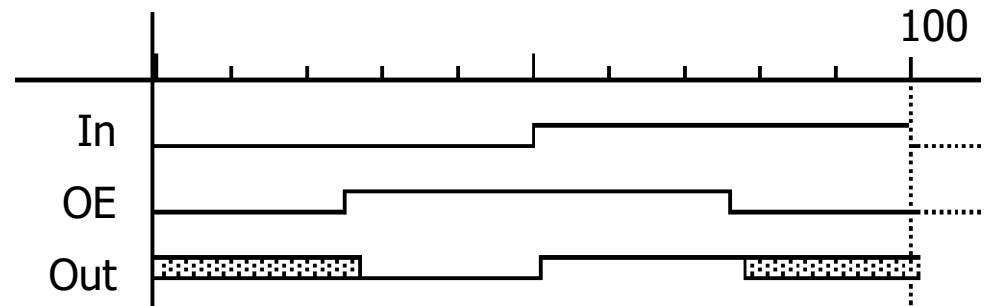


# Tri-state gates

- The third value
  - ◆ logic values: “0”, “1”
  - ◆ don't care: “X” (must be 0 or 1 in real circuit!)
  - ◆ third value or state: “Z” — high impedance, infinite R, no connection
- Tri-state gates
  - ◆ additional input – output enable (OE)
  - ◆ output values are 0, 1, and Z
  - ◆ when OE is high, the gate functions normally
  - ◆ when OE is low, the gate is disconnected from wire at output
  - ◆ allows more than one gate to be connected to the same output wire
    - as long as only one has its output enabled at any one time (otherwise, sparks could fly)

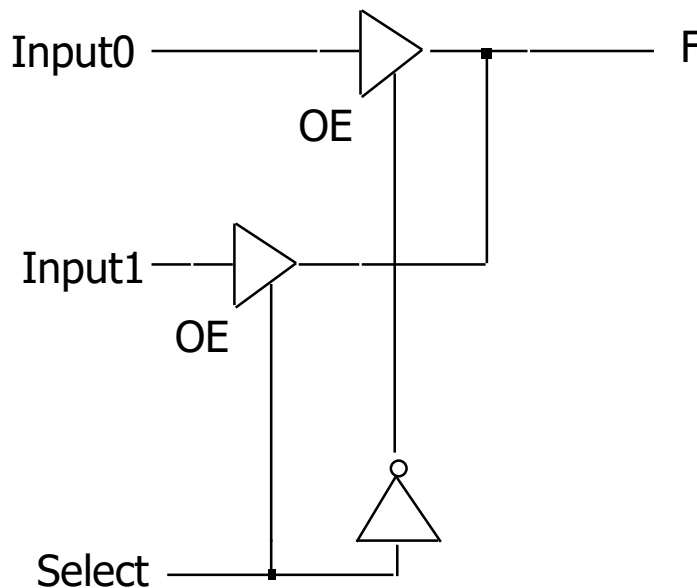


non-inverting tri-state buffer	In	OE	Out
	X	0	Z
	0	1	0
	1	1	1



# Tri-state and multiplexing

- When using tri-state logic
  - ◆ (1) make sure never more than one "driver" for a wire at any one time (pulling high and low at the same time can severely damage circuits)
  - ◆ (2) make sure to only use value on wire when its being driven (using a floating value may cause failures)
- Using tri-state gates to implement an economical multiplexer



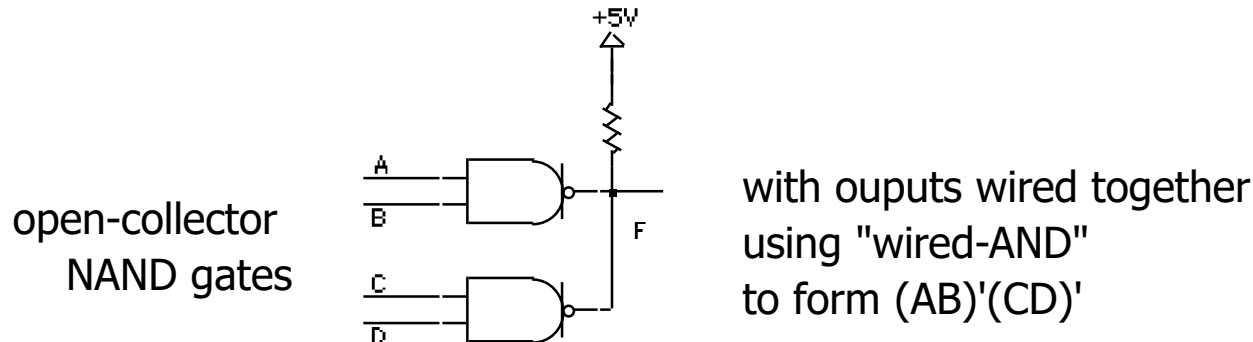
when Select is high  
Input1 is connected to F

when Select is low  
Input0 is connected to F

this is essentially a 2:1 mux

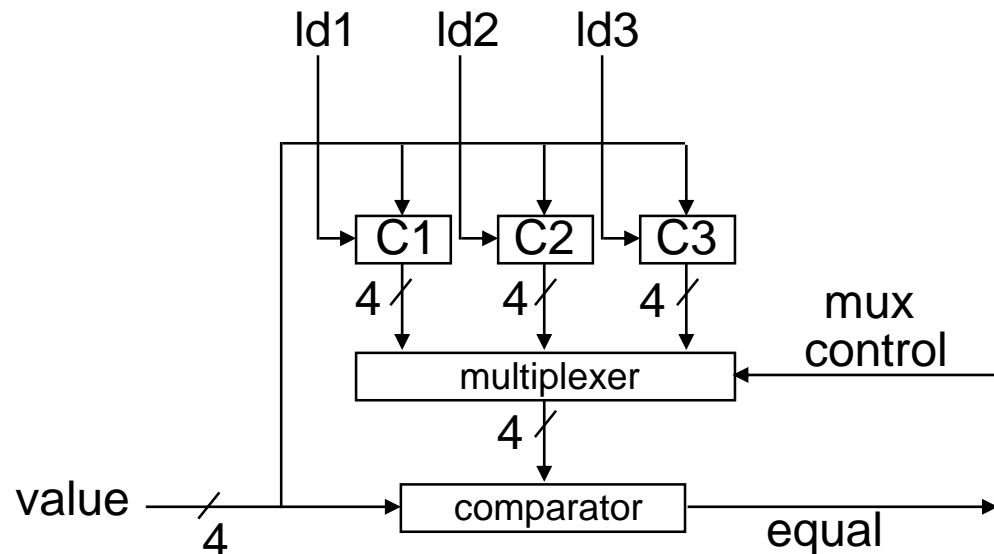
# Open-collector gates and wired-AND

- Open collector: another way to connect gate outputs to the same wire
  - ◆ gate only has the ability to pull its output low
  - ◆ it cannot actively drive the wire high (default – pulled high through resistor)
- Wired-AND can be implemented with open collector logic
  - ◆ if A and B are "1", output is actively pulled low
  - ◆ if C and D are "1", output is actively pulled low
  - ◆ if one gate output is low and the other high, then low wins
  - ◆ if both gate outputs are "1", the wire value "floats", pulled high by resistor
    - low to high transition usually slower than it would have been with a gate pulling high
  - ◆ hence, the two NAND functions are ANDed together



# Digital Combination Lock New DP

- Decrease number of inputs
- Remove 3 code digits as inputs
  - ◆ use code registers
  - ◆ make them loadable from value
  - ◆ need 3 load signal inputs (net gain in input  $(4 \times 3) - 3 = 9$ )
    - could be done with 2 signals and decoder (ld1, ld2, ld3, load none)



# Serial Line Transmitter/Receiver

## ■ Problem specification

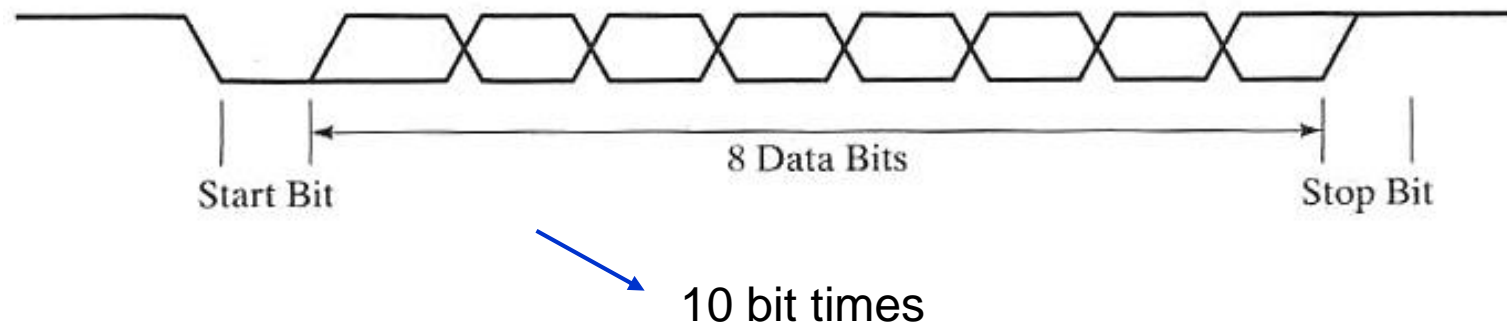
### ◆ Transmitter

- take input from a telephone-like keypad
- send a byte corresponding to the key that was pressed over a single wire one bit at a time

### ◆ Receiver

- receive the serial data sent by the first
- display it on a small LCD screen

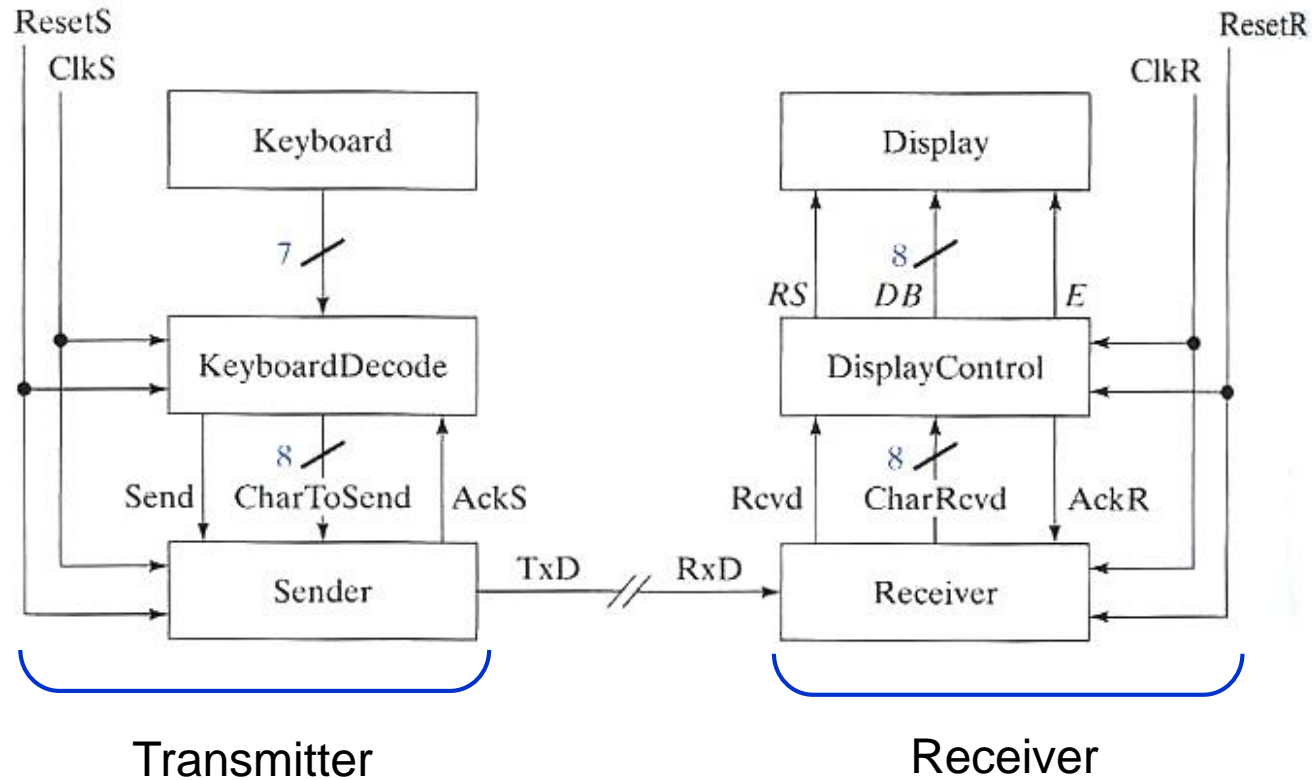
### ◆ Using RS-232 protocol for formatting the data on the wire





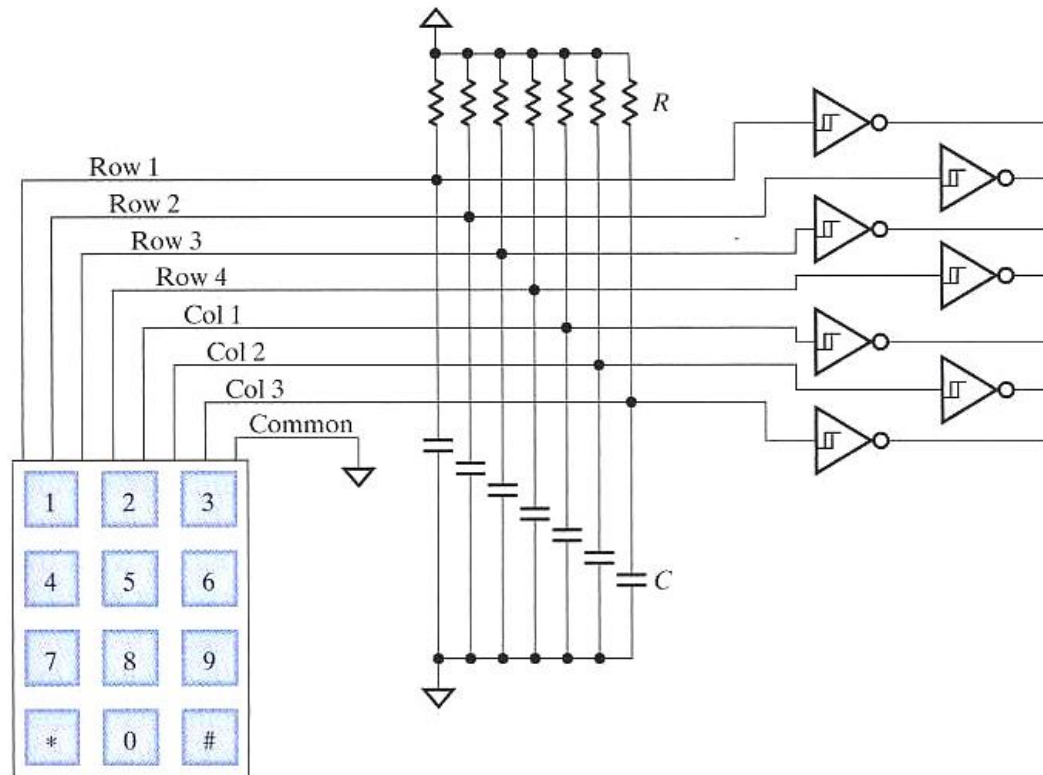
# Serial Line Transmitter/Receiver

- Block diagram



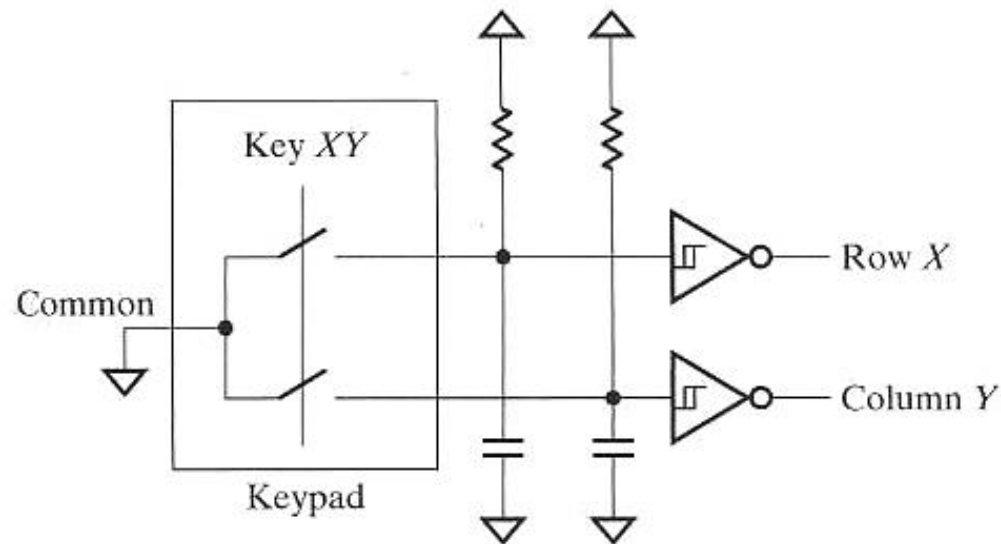
# Serial Line Transmitter/Receiver

- Keyboard
  - ◆ Keypad and Debouncing circuitry
    - With this arrangement, the corresponding row and column wires at the inverter outputs go high when a key is pressed



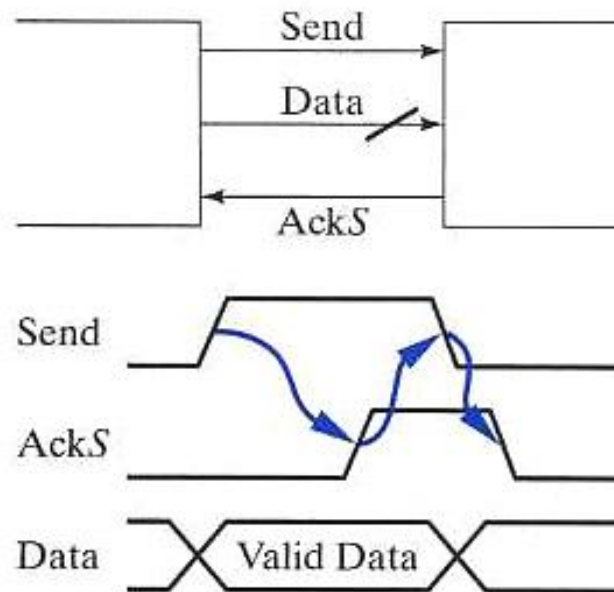
# Serial Line Transmitter/Receiver

- Keyboard (cont.)
  - ◆ Schematic diagram of a single keyboard key



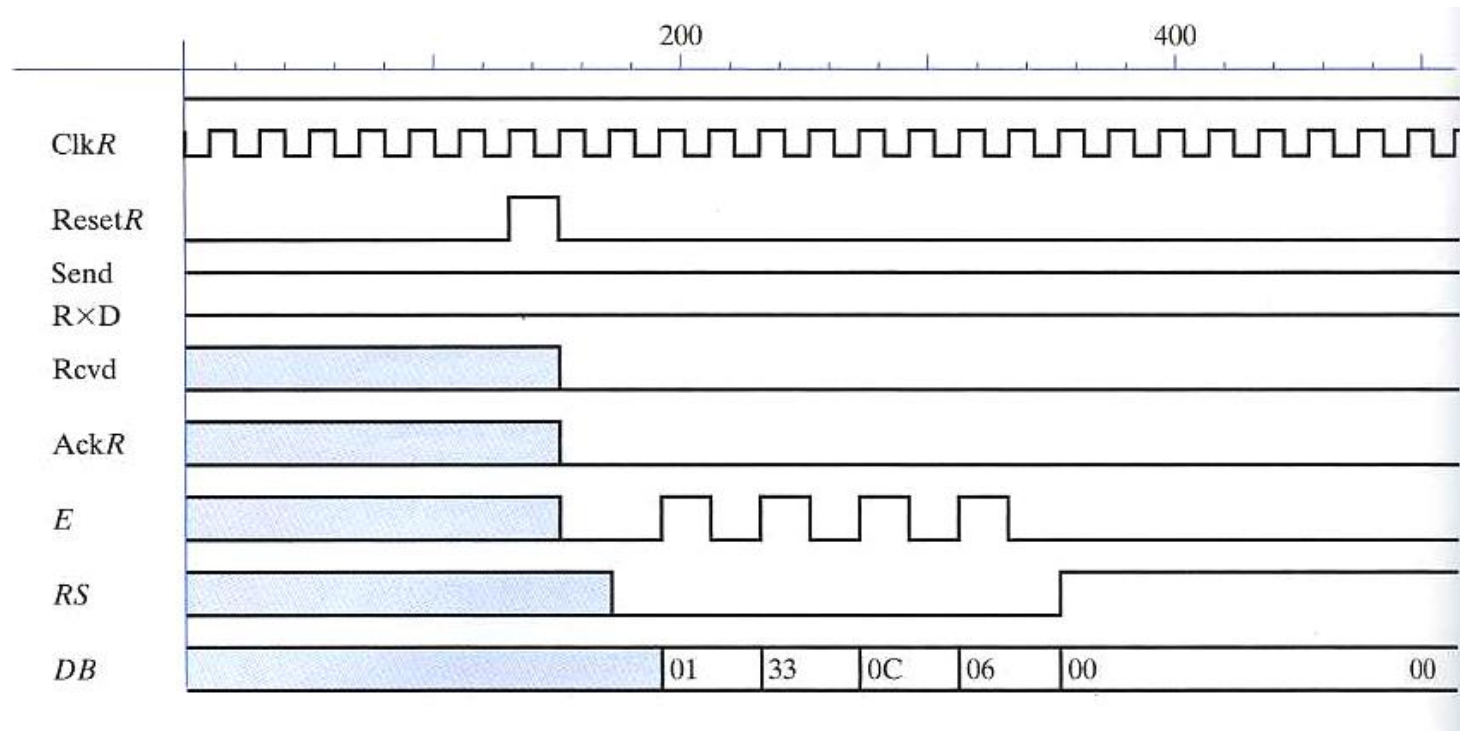
# Serial Line Transmitter/Receiver

- Keyboard decode
  - ◆ include **four-cycle handshake** between the *KeyboardDecode* block and the *Sender* block



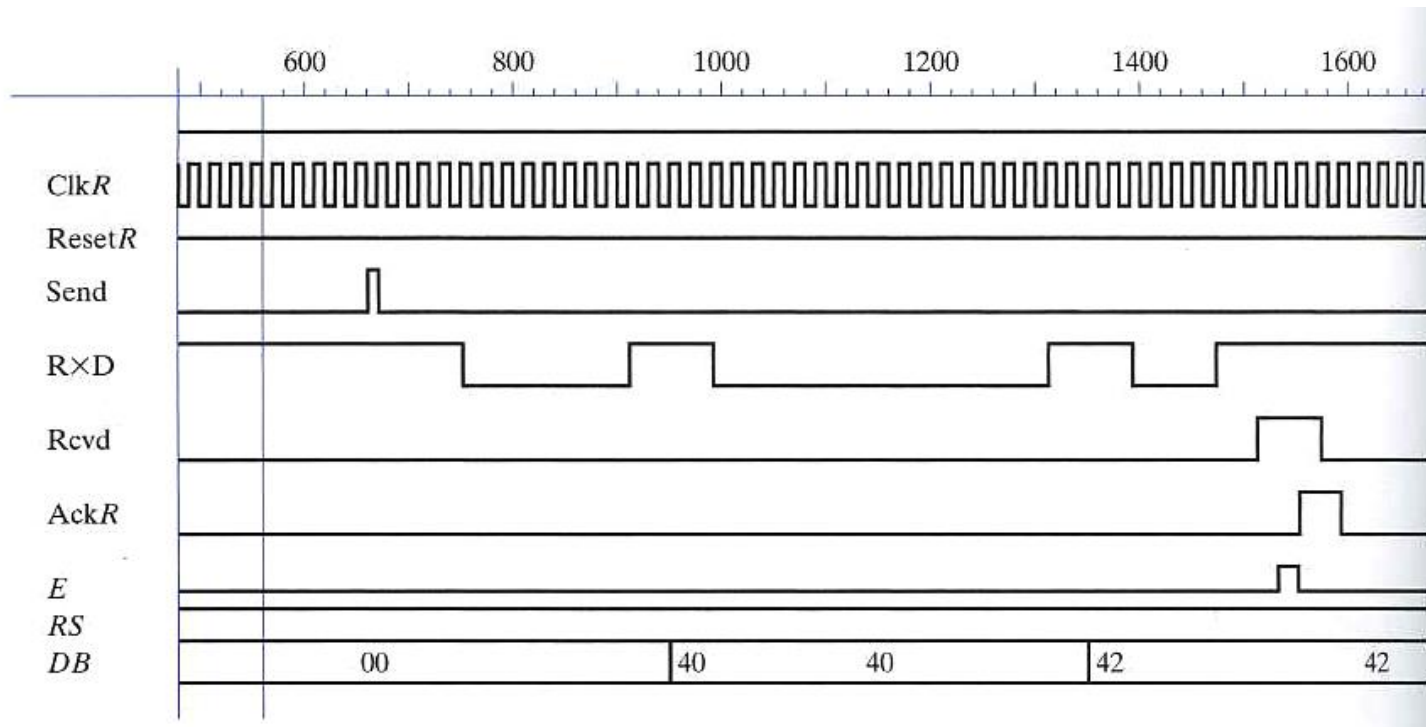
# Serial Line Transmitter/Receiver

- Timing diagram for the receiver half
  - ◆ Initialization sequence of the *DisplayControl* module



# Serial Line Transmitter/Receiver

- Timing diagram for the receiver half (cont.)
  - ◆ *Receiver* receiving a character and the handshake with the *DisplayControl* module to get it on the LCD



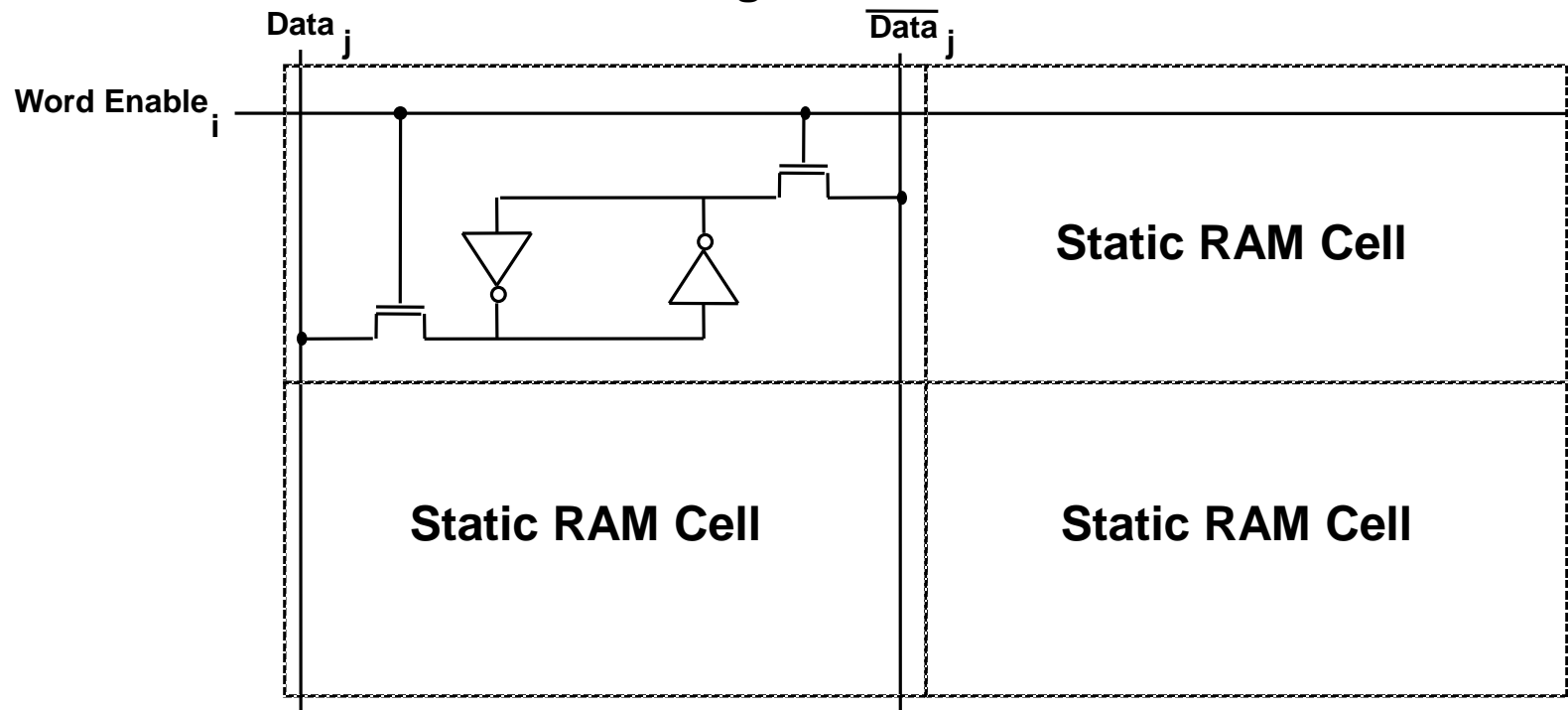
# Random Access Memories (SRAM)

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

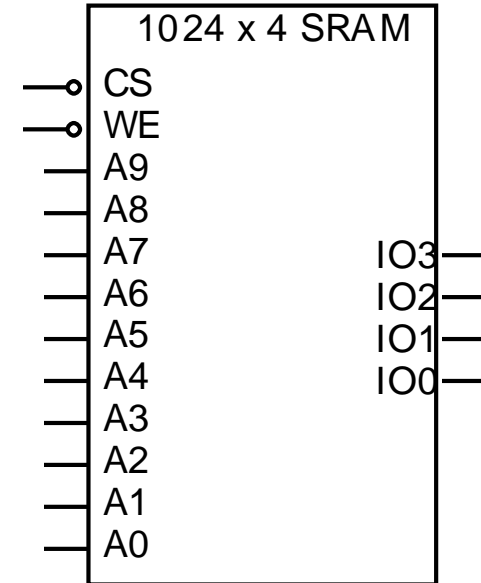
## ■ *Static RAM Organization*

**Chip Select Line (active lo)**

**Write Enable Line (active lo)**

**10 Address Lines**

**4 Bidirectional Data Lines**

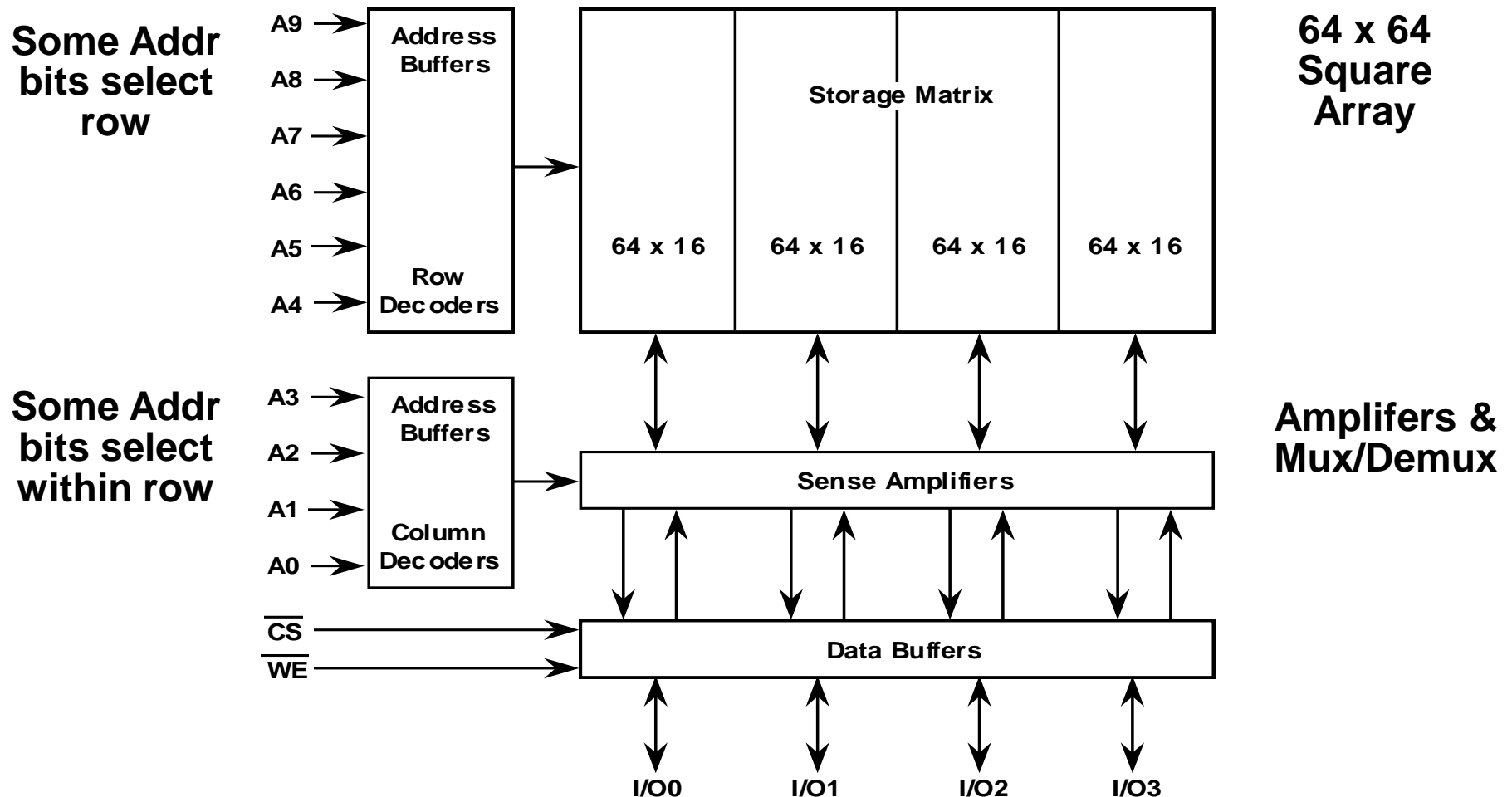




# Random Access Memories

## RAM Organization

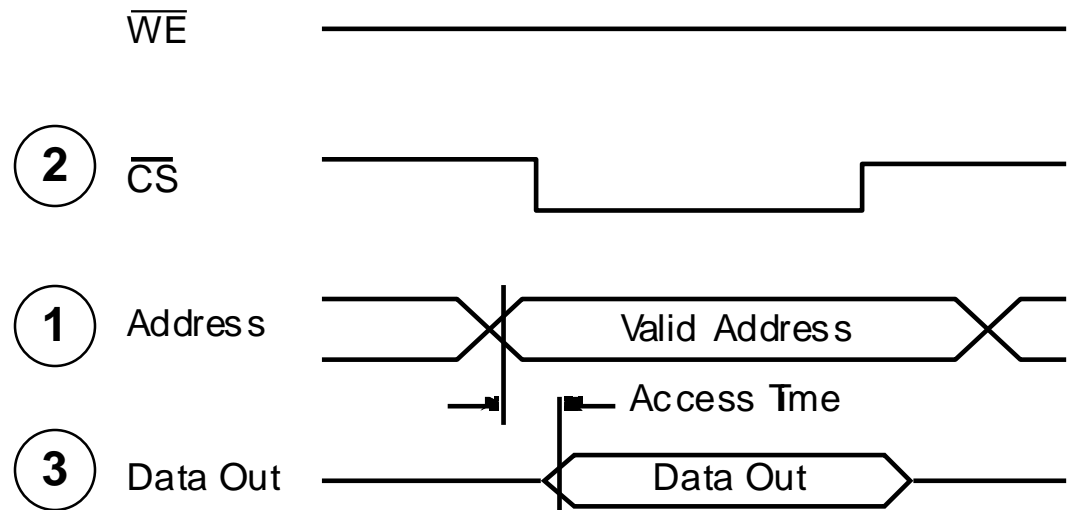
Long thin layouts are not the best organization for a RAM



# Random Access Memories

## RAM Timing

### Simplified Read Timing

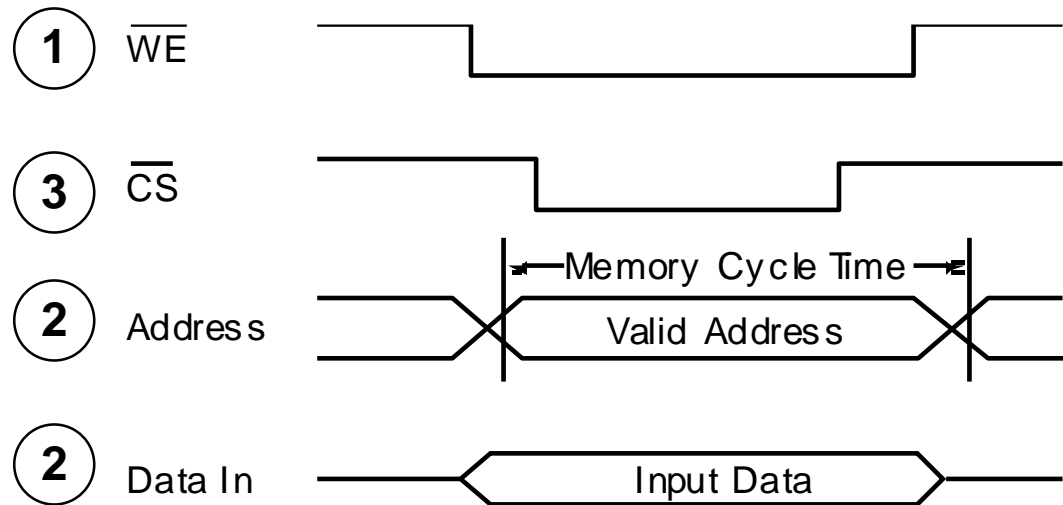


- 1) valid address must be set up on the address line
- 2)  $\overline{CS}$  line is taken low while  $\overline{WE}$  stays high
- 3) access time is the time it takes for new data to be ready to appear at the output driver

# Random Access Memories

## Write operation

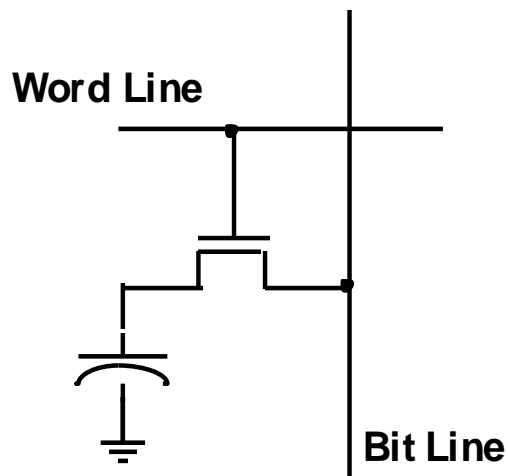
### Simplified Write Timing



- 1)  $\overline{WE}$  should be brought low
- 2) address and data lines should be stable
- 3)  $\overline{CS}$  line goes low

# Random Access Memories

## *Dynamic RAMs*



**1 Transistor (+ capacitor) memory element**

**Read: Assert Word Line, Sense Bit Line**

**Write: Drive Bit Line, Assert Word Line**

**<< Problem >>**

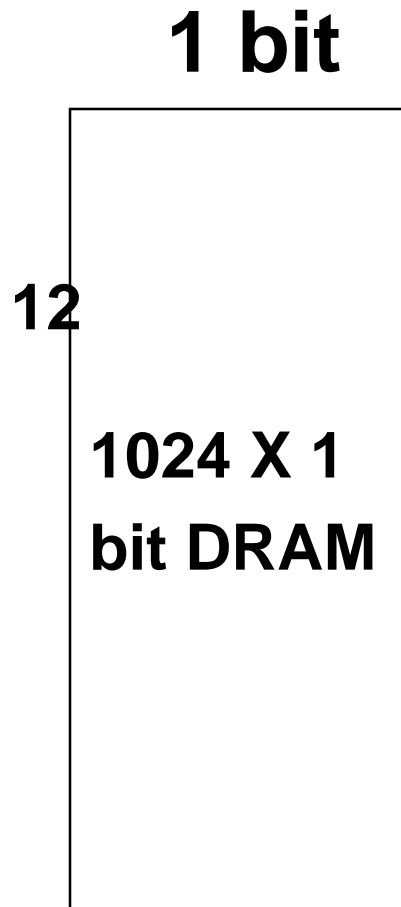
**1) Destructive Read-Out : discharge**  
(write after read operation)

**2) Need for Refresh Cycles: storage decay in ms**  
(periodically, memory elements must be read and written back to their locations)

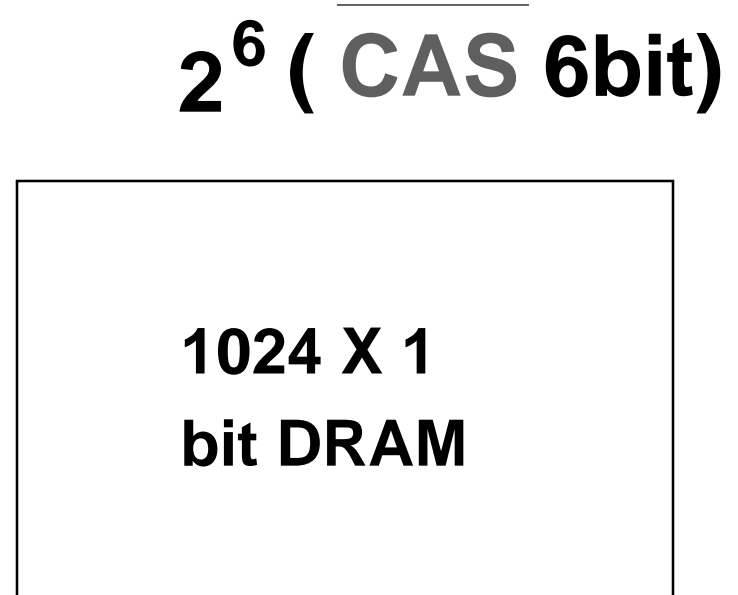
**Internal circuits read word and write back**

# Random Access Memories

- 1024 X 1 bit DRAM



**$2^6$**   
**(  $\overline{\text{RAS}}$  6bit)**



**RAS: Row Address Strobe (first)**

**CAS :Column Address Strobe (second)**

# Random Access Memories

## *DRAM Organization*

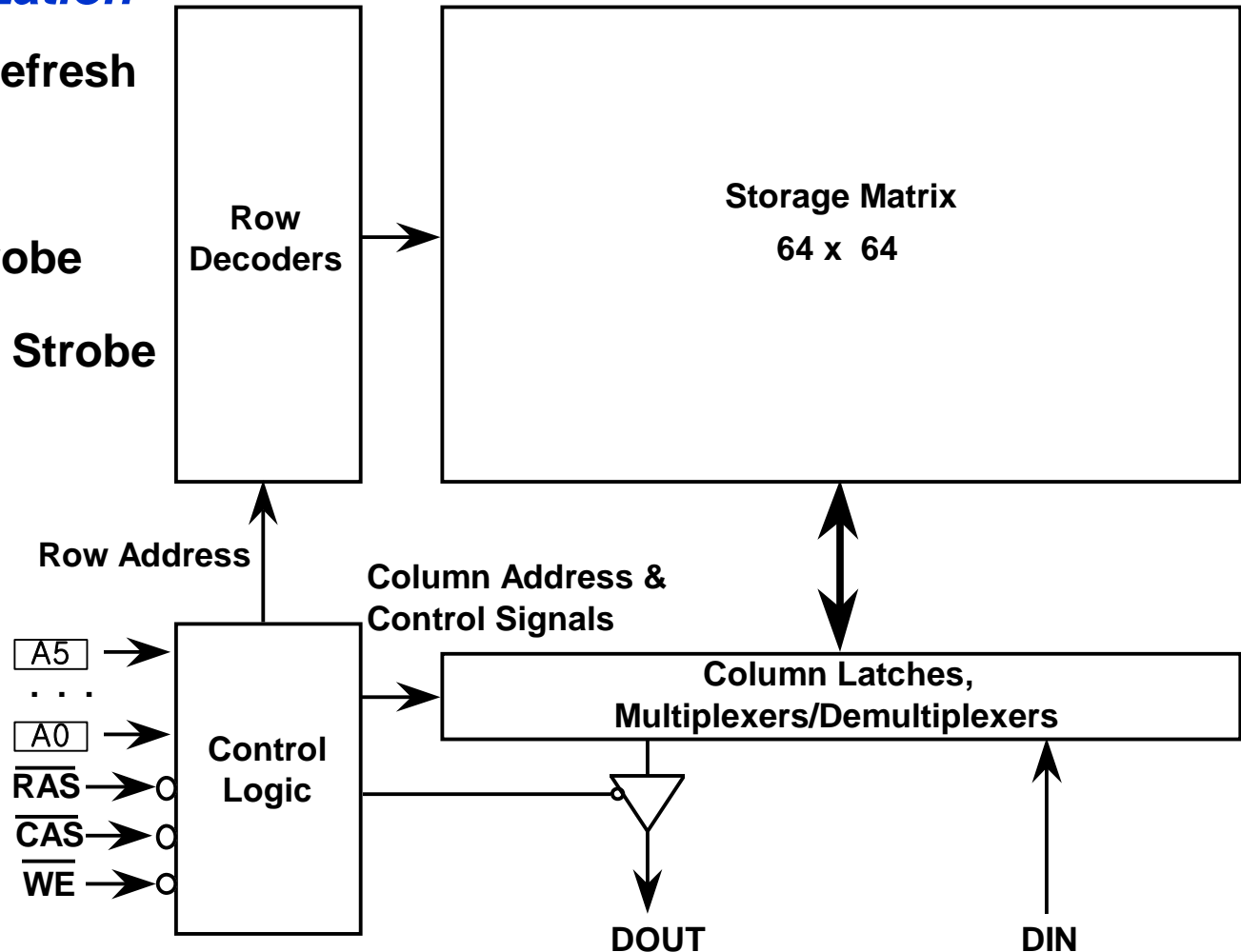
Long rows to simplify refresh

Two new signals: ,

**RAS: Row Address Strobe**

**CAS :Column Address Strobe**

replace Chip Select



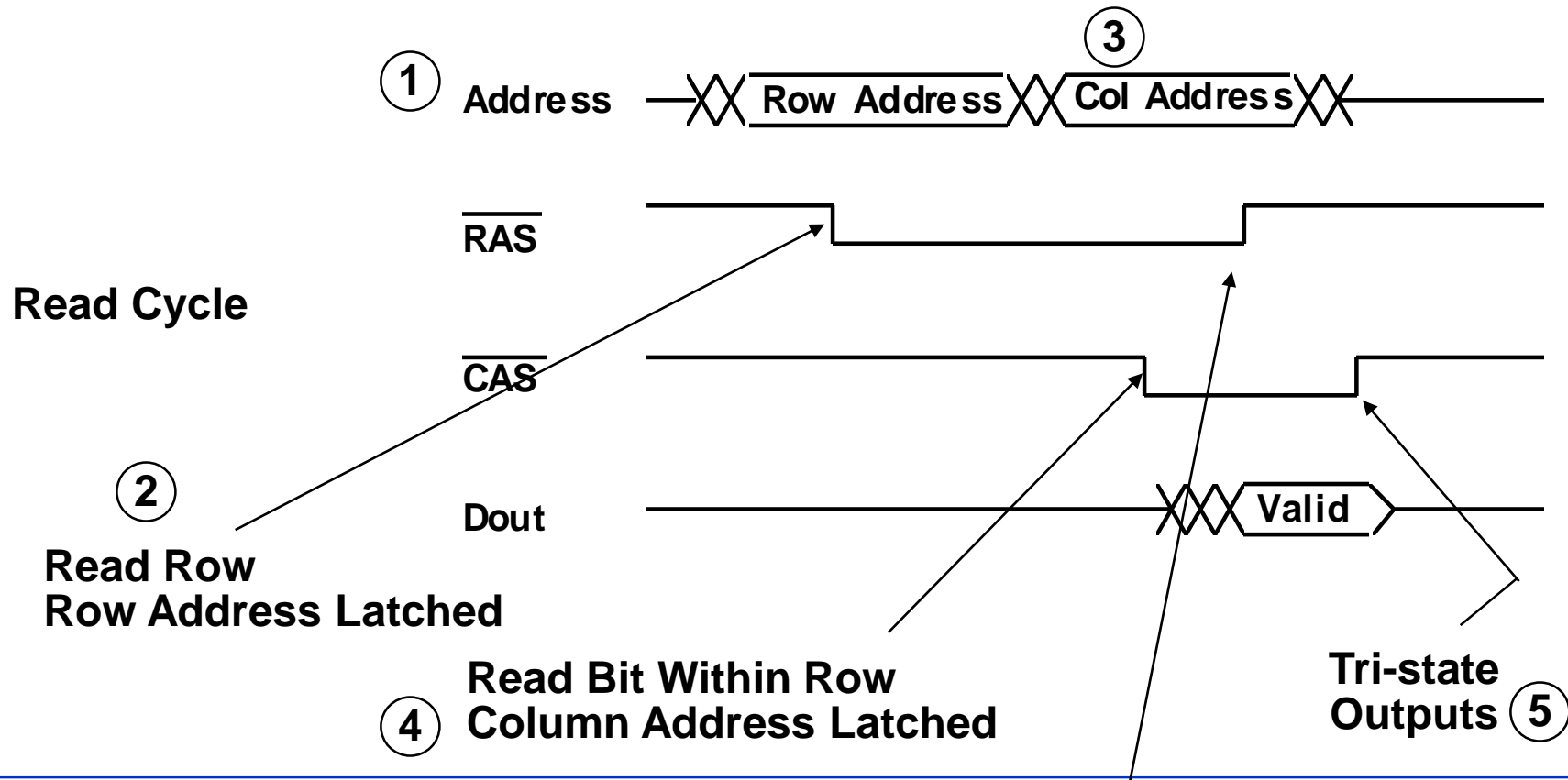
*1024 by 1 bit DRAM block diagram*

# Random Access Memories

## *RAS, CAS Addressing*

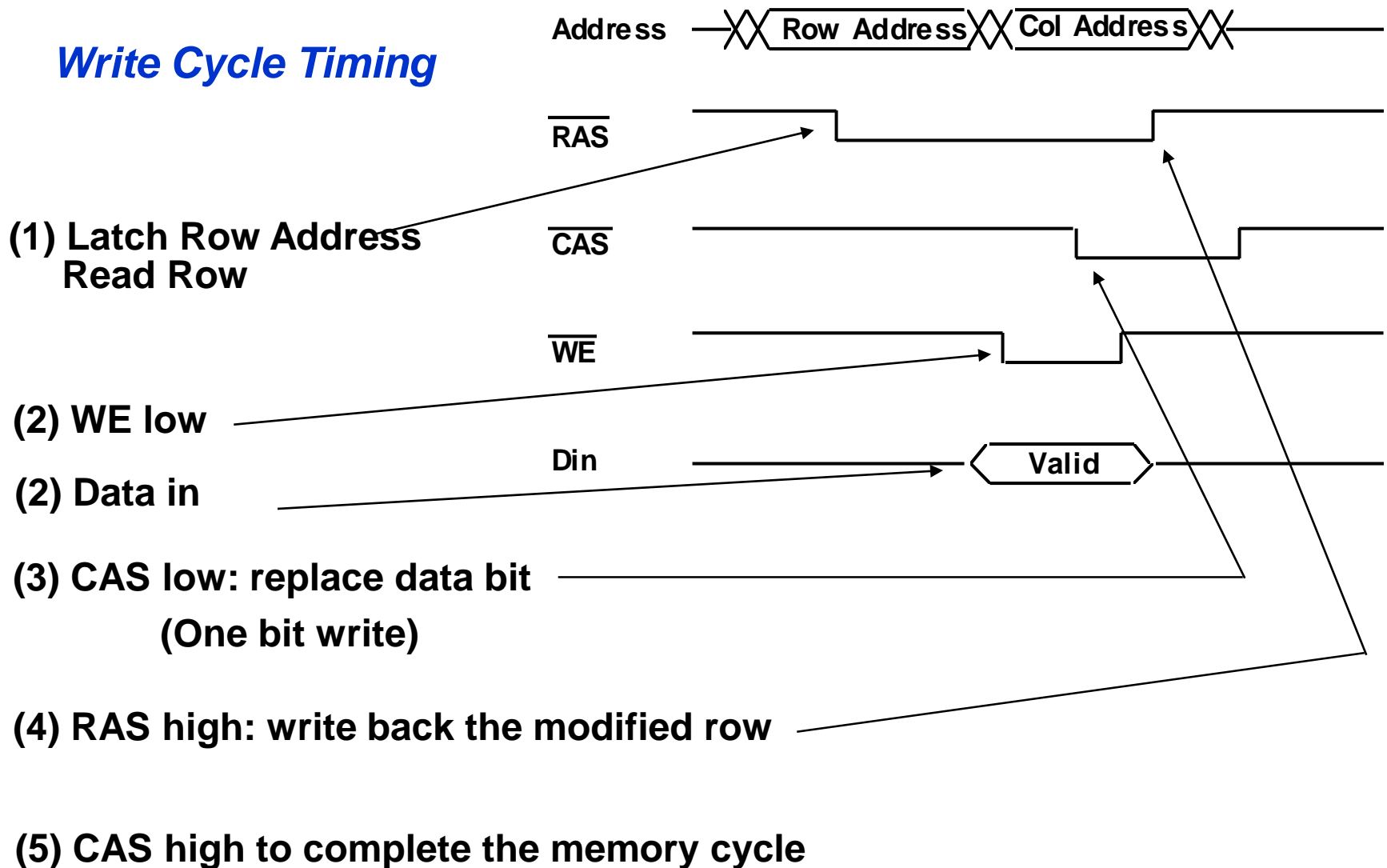
Even to read 1 bit, an entire 64-bit row is read!

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



# Random Access Memories

## Write Cycle Timing





# Section Summary

---

- FSM design
  - ◆ understanding the problem
  - ◆ generating state diagram
  - ◆ communicating state machines
- Four case studies
  - ◆ understand I/O behavior
  - ◆ draw diagrams
  - ◆ enumerate states for the "goal"
  - ◆ expand with error conditions
  - ◆ reuse states whenever possible