EECS 151/251A
Fall 2018
Digital Design and Integrated Circuits

Instructor:
John Wawrzynek & Nicholas Weaver

Lecture 6

Boolean Algebra

Boolean Algebra

Set of elements B, binary operators $\{+,\bullet\}$, unary operation $\{'\}$, such that the following axioms hold:

1. B contains at least two elements a,b such that $a \neq b$.
2. Closure: a,b in B,

   $a+b$ in B, $a\bullet b$ in B, a' in B.
3. Communitive laws:

   $a+b=b+a, \quad a\bullet b=b\bullet a.$
4. Identities: $0, 1$ in B

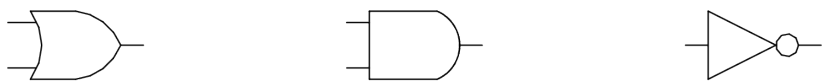
   $a+0=a, \quad a\bullet 1=a.$
5. Distributive laws:

   $a+(b\bullet c)=(a+b)\bullet(a+c), \; a\bullet(b+c)=a\bullet b+a\bullet c.$
6. Complement:

   $a+a'=1, \; a\bullet a'=0.$

$B=\{0,1\}$, $+=$ OR, $\bullet=$ AND, $'=$ NOT

is a valid Boolean Algebra.



00	0
01	1
10	1
11	1

00	0
01	0
10	0
11	1

0	1
1	0

Some Laws of Boolean Algebra

Duality: A dual of a Boolean expression is derived by interchanging OR and AND operations, and 0s and 1s (literals are left unchanged).

$$\{F(x_1, x_2, \ldots, x_n, 0, 1, +, \bullet)\}^D = \{F(x_1, x_2, \ldots, x_n, 1, 0, \bullet, +)\}$$

Any law that is true for an expression is also true for its dual.

Operations with 0 and 1:

```
x + 0 = x          x * 1 = x
x + 1 = 1          x * 0 = 0
```

Idempotent Law:

```
x + x = x          x * x = x x = x
```

Involution Law:

```
(x')' = x
```

Laws of Complementarity:

```
x + x' = 1         x x' = 0
```

Commutative Law:

```
x + y = y + x      x y = y x
```

Some Laws of Boolean Algebra (cont.)

Associative Laws:

```
(x + y) + z = x + (y + z)
x y z = x(y z)
```

Distributive Laws:

```
x (y + z) = (x y) + (x z)
x +(y z) = (x + y)(x + z)
```

“Simplification” Theorems:

```
x y + x y' = x
(x + y) (x + y') = x
x + x y = x
x (x + y) = x
```

DeMorgan's Law:

```
(x + y + z + …)' = x'y'z'
(x y z …)' = x' + y' +z'
```

Theorem for Multiplying and Factoring:

```
(x + y) (x' + z) = x z + x'y
```

Consensus Theorem:

```
x y + y z + x' z = (x + y) (y + z) (x' + z)
x y + x' z = (x + y) (x' + z)
```

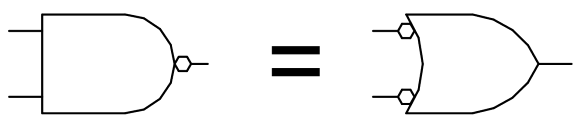
DeMorgan's Law

$$(x + y)' = x' y'$$

Exhaustive Proof

x	y	x'	y'	(x + y)'	x'y'
0	0	1	1	1	1
0	1	1	0	0	0
1	0	0	1	0	0
1	1	0	0	0	0

$$(x\, y)' = x' + y'$$

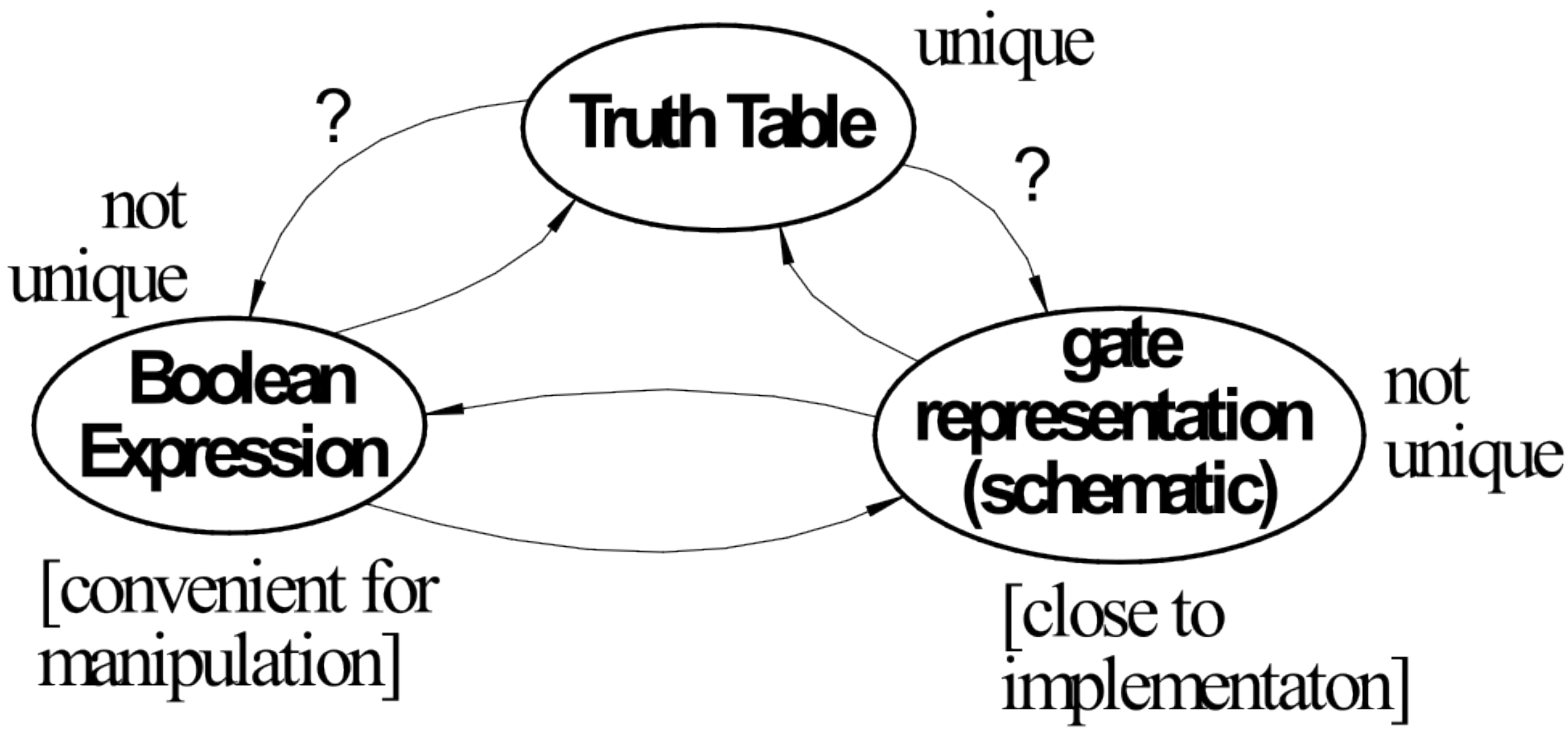


Exhaustive Proof

x	y	x'	y'	(x y)'	x' + y'
0	0	1	1	1	1
0	1	1	0	1	1
1	0	0	1	1	1
1	1	0	0	0	0

Relationship Among Representations

* Theorem: Any Boolean function that can be expressed as a truth table can be written as an expression in Boolean Algebra using AND, OR, NOT.



How do we convert from one to the other?

Canonical Forms

- Standard form for a Boolean expression - unique algebraic expression directly from a true table (TT) description.
- Two Types:
  * **Sum of Products (SOP)**
  * **Product of Sums (POS)**
- Sum of Products (disjunctive normal form, minterm expansion). Example:

Minterms	a	b	c	f	f'
a'b'c'	0	0	0	0	1
a'b'c'	0	0	1	0	1
a'bc'	0	1	0	0	1
a'bc	0	1	1	1	0
ab'c'	1	0	0	1	0
ab'c	1	0	1	1	0
abc'	1	1	0	1	0
abc	1	1	1	1	0

One product (and) term for each 1 in f:

f = a'bc + ab'c' + ab'c + abc' + abc

f' = a'b'c' + a'b'c + a'bc'

What is the cost?

Sum of Products (cont.)

Canonical Forms are usually not minimal:

Our Example:

$$
\begin{aligned}
f &= a'bc + ab'c' + ab'c + abc' + abc && (xy' + xy = x) \\
&= a'bc + ab' + ab \\
&= a'bc + a && (x'y + x = y + x) \\
&= a + bc
\end{aligned}
$$

$$
\begin{aligned}
f' &= a'b'c' + a'b'c + a'bc' \\
&= a'b' + a'bc' \\
&= a'(b' + bc') \\
&= a'(b' + c') \\
&= a'b' + a'c'
\end{aligned}
$$

Canonical Forms

- Product of Sums (conjunctive normal form, maxterm expansion).

Example:

maxterms	a	b	c	f	f'
a+b+c	0	0	0	0	1
a+b+c'	0	0	1	0	1
a+b'+c	0	1	0	0	1
a+b'+c'	0	1	1	1	0
a'+b+c	1	0	0	1	0
a'+b+c'	1	0	1	1	0
a'+b'+c	1	1	0	1	0
a'+b'+c'	1	1	1	1	0

*One sum (**or**) term for each **0** in f:*

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

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

Simplify … algebra or K-maps

- Algebra: `f = a+bc`
- K-maps:

c \ ab	00	01	11	10
0	0	0	1	1
1	0	1	1	1

Multi-level Logic

Multi-level Combinational Logic

- Example: reduced sum-of-products form

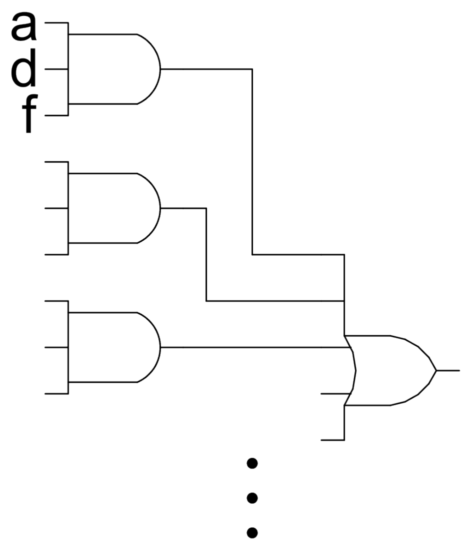
  `x = adf + aef + bdf + bef + cdf + cef + g`

- Implementation in 2-levels with gates:

  <u>cost:</u> 1 7-input OR, 6 3-input AND

  => 50 transistors

  <u>delay:</u> 3-input OR gate delay + 7-input AND gate delay



Multi-level Combinational Logic

- Example: reduced sum-of-products form

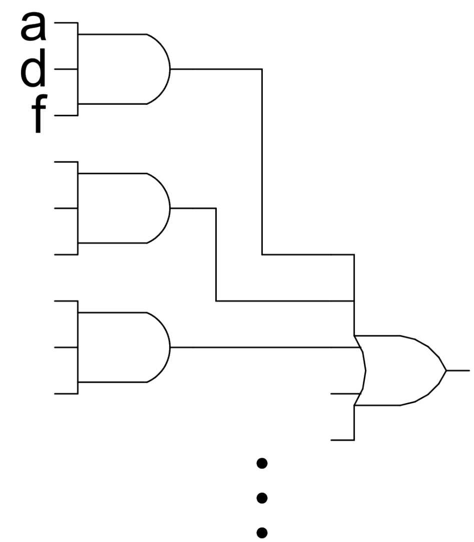
  `x = adf + aef + bdf + bef + cdf + cef + g`

- Implementation in 2-levels with gates:

  cost: 1 7-input OR, 6 3-input AND

  => 50 transistors

  delay: 3-input OR gate delay + 7-input AND gate delay



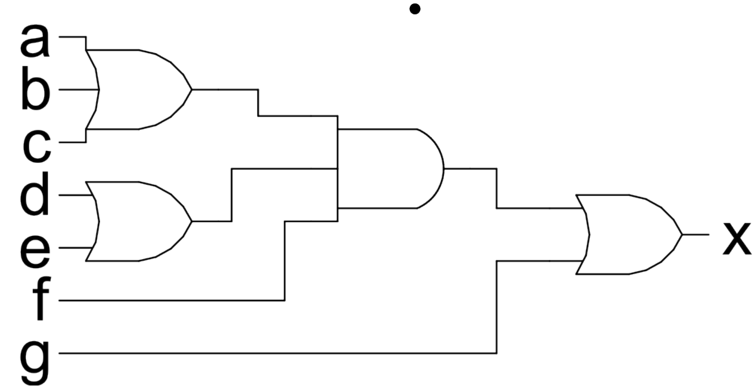
- Factored form:

  `x = (a + b +c)(d + e)f + g`

  cost: 1 3-input OR, 2 2-input OR, 1 3-input AND

  => 20 transistors

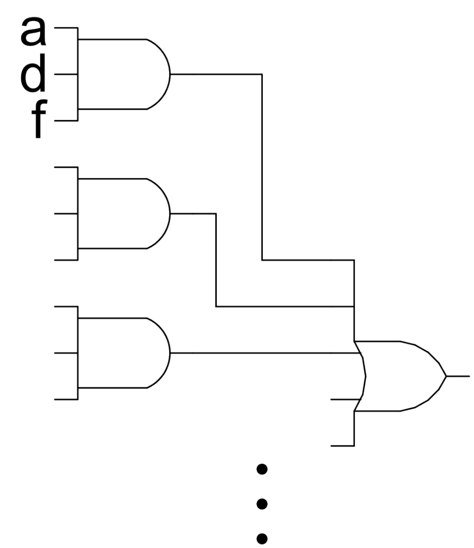
  delay: 3-input OR + 3-input AND + 2-input OR



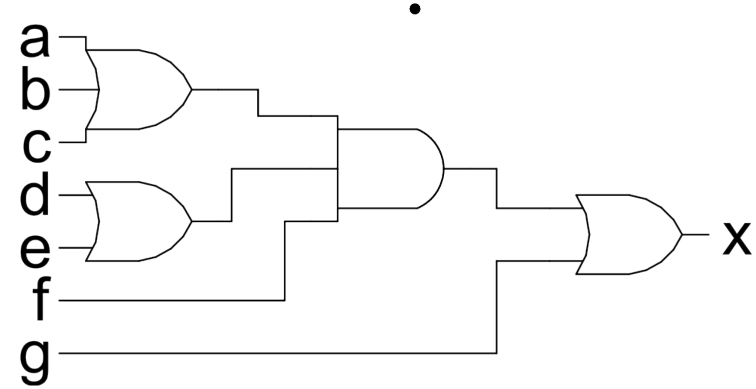
Footnote: NAND would be used in place of all ANDs and ORs.

Multi-level Combinational Logic

- Example: reduced sum-of-products form
  `x = adf + aef + bdf + bef + cdf + cef + g`
- Implementation in 2-levels with gates:
  cost: 1 7-input OR, 6 3-input AND
  => 50 transistors
  delay: 3-input OR gate delay + 7-input AND gate delay



- Factored form:
  `x = (a + b +c)(d + e)f + g`
  cost: 1 3-input OR, 2 2-input OR, 1 3-input AND
  => 20 transistors
  delay: 3-input OR + 3-input AND + 2-input OR



Footnote: NAND would be used in place of all ANDs and ORs.

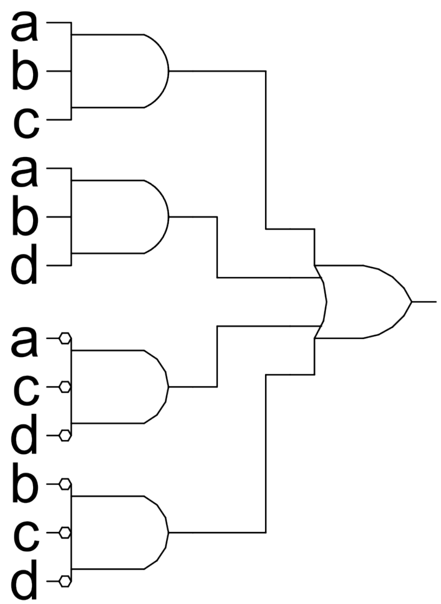
Which is faster?

In general: Using multiple levels (more than 2) will reduce the cost. Sometimes also delay. Sometimes a tradeoff between cost and delay.

In reality: The ASIC mapping tools target cells are optimized: the tool will decide

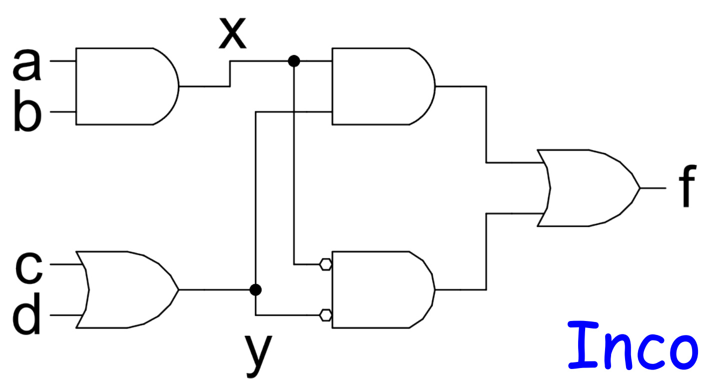
Multi-level Combinational Logic

Another Example: $F = abc + abd + a'c'd' + b'c'd'$



let $x = ab$ $\quad y = c+d$

$$f = xy + x'y'$$



Incorporates fanout.

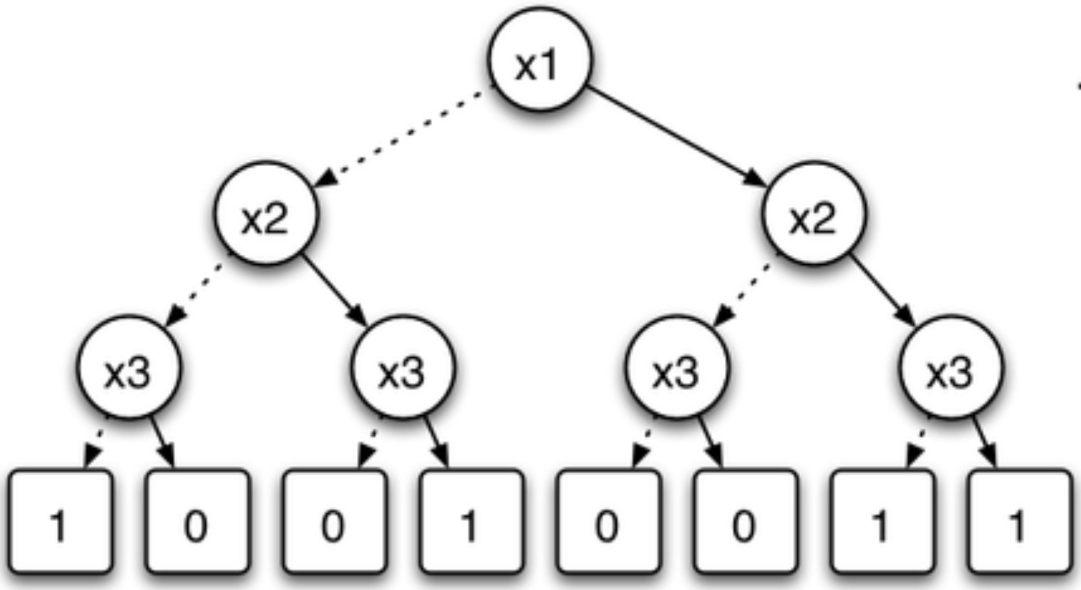
No convenient hand methods exist for multi-level logic simplification:

a) CAD Tools use sophisticated algorithms and heuristics

Guess what? These problems tend to be NP-complete

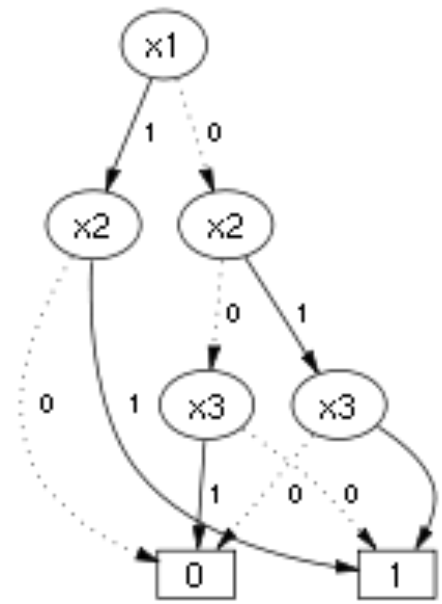
b) Humans and tools often exploit some special structure (example adder)

Binary decision diagrams as the base



Binary decision tree

x1	x2	x3	*f*
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1



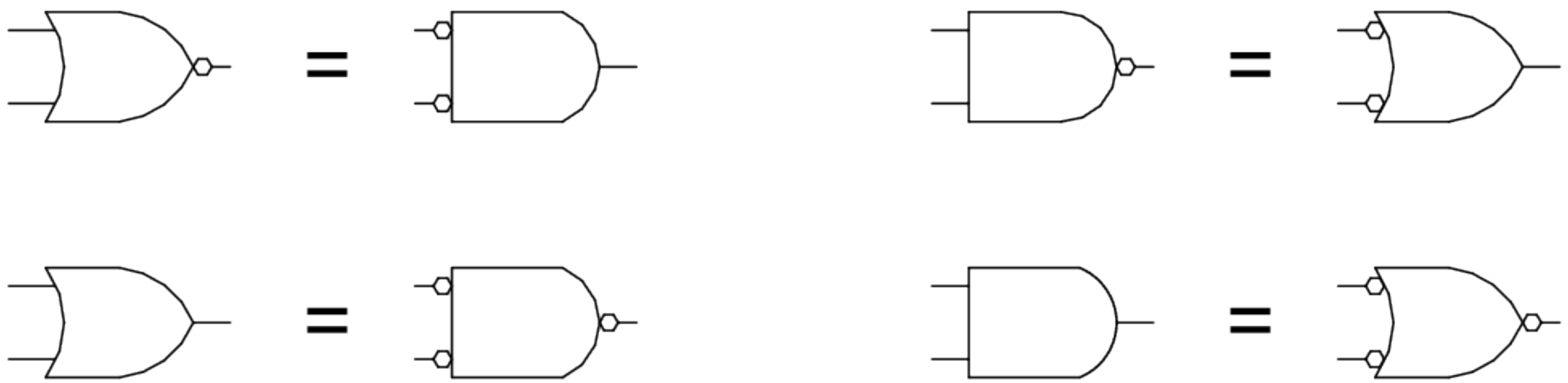
Binary decision diagram (BDD)

NAND-NAND & NOR-NOR Networks

DeMorgan's Law Review:

$(a + b)' = a'\,b'$ $\qquad$ $(a\,b)' = a' + b'$

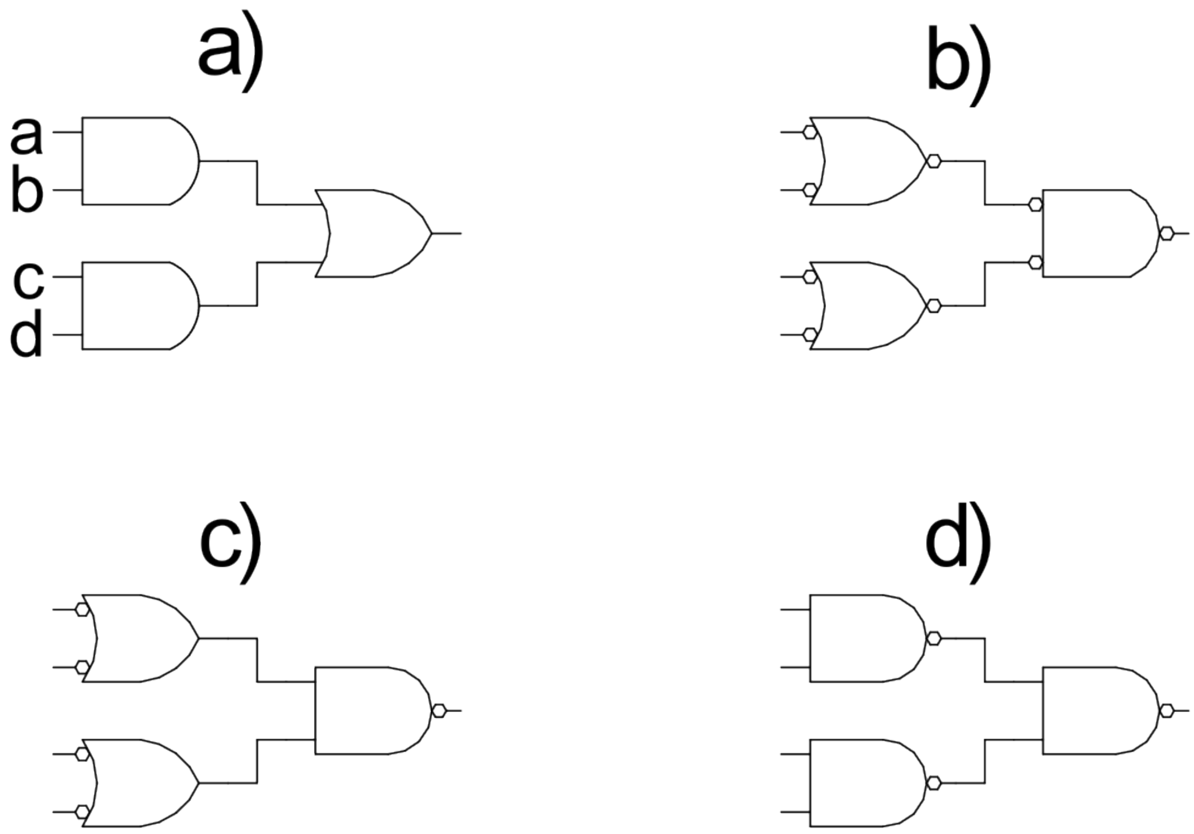
$a + b = (a'\,b')'$ $\qquad$ $(a\,b) = (a' + b')'$



push bubbles or *introduce in pairs* or *remove pairs:* $(x')' = x$.

NAND-NAND & NOR-NOR Networks

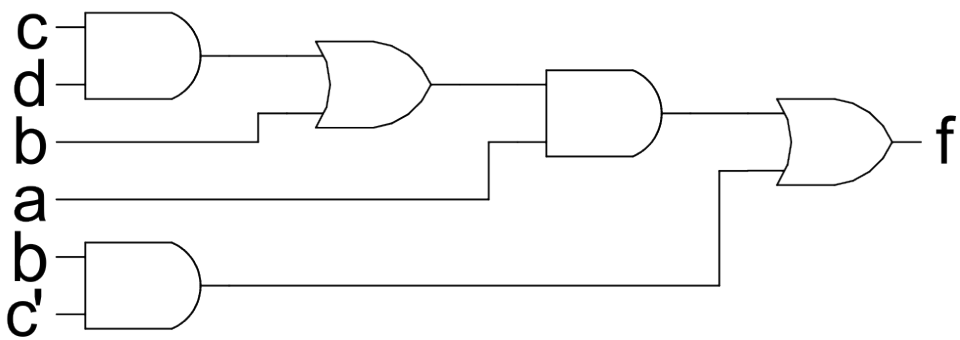
- Mapping from AND/OR to NAND/NAND



Multi-level Networks

Convert to NANDs:

F = a(b + cd) + bc'



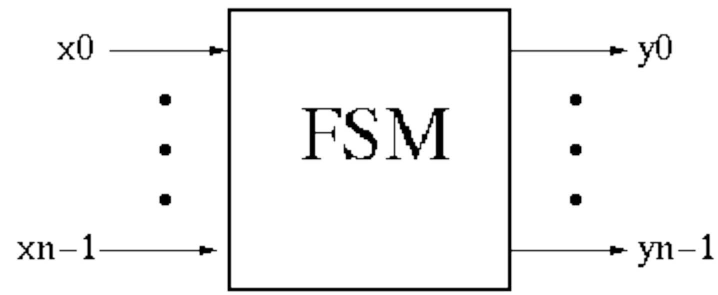
NAN

PRETTY STATE MACHINE

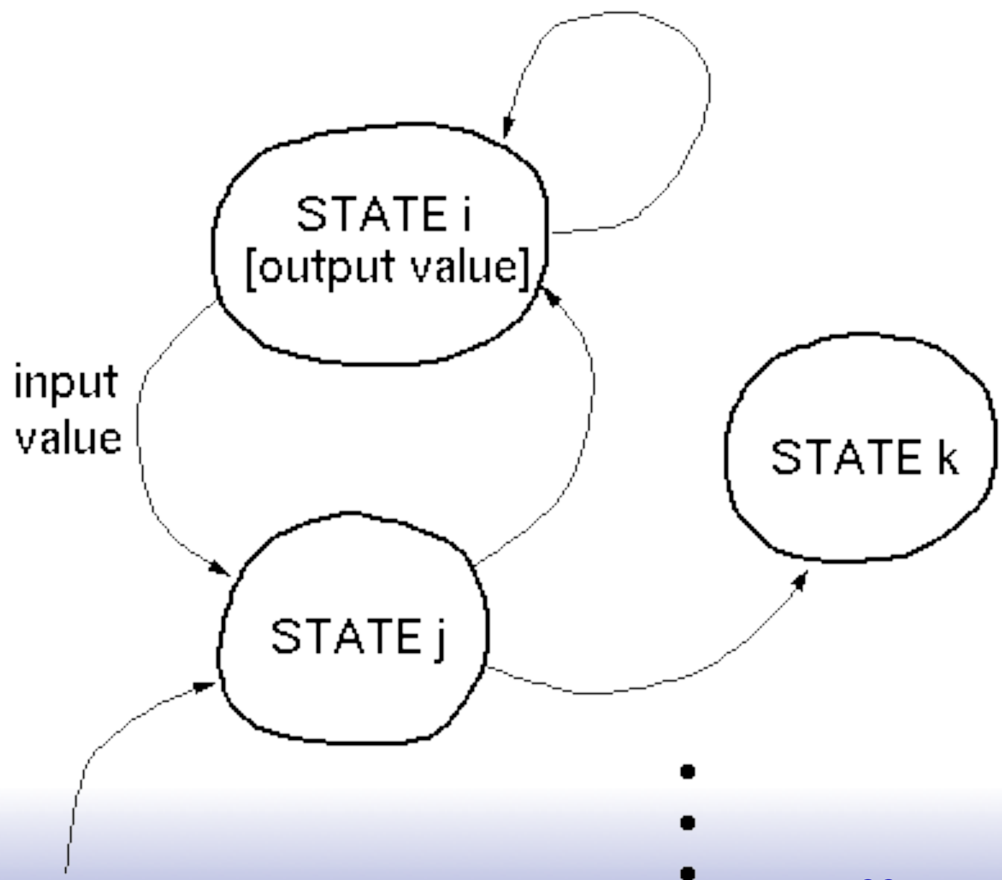
Finite State Machines

Finite State Machines (FSMs)

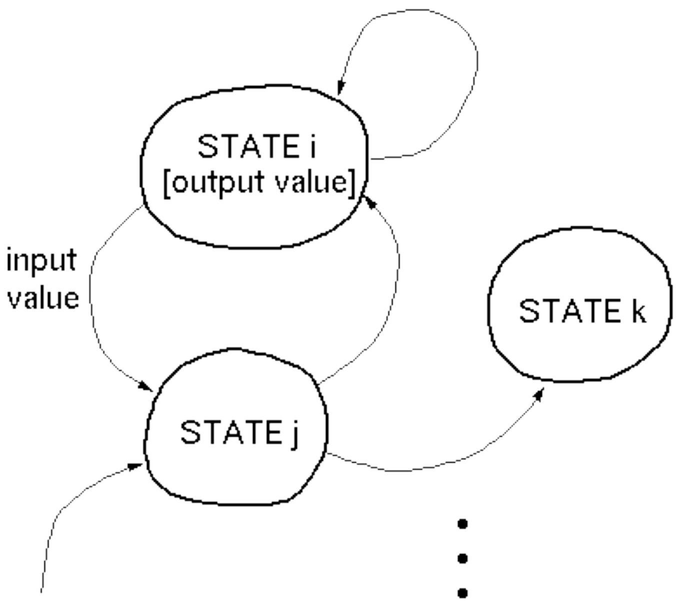
- **FSM** circuits are a type of *sequential circuit*:
  - output depends on present *and* past inputs
    - effect of past inputs is represented by the current *state*

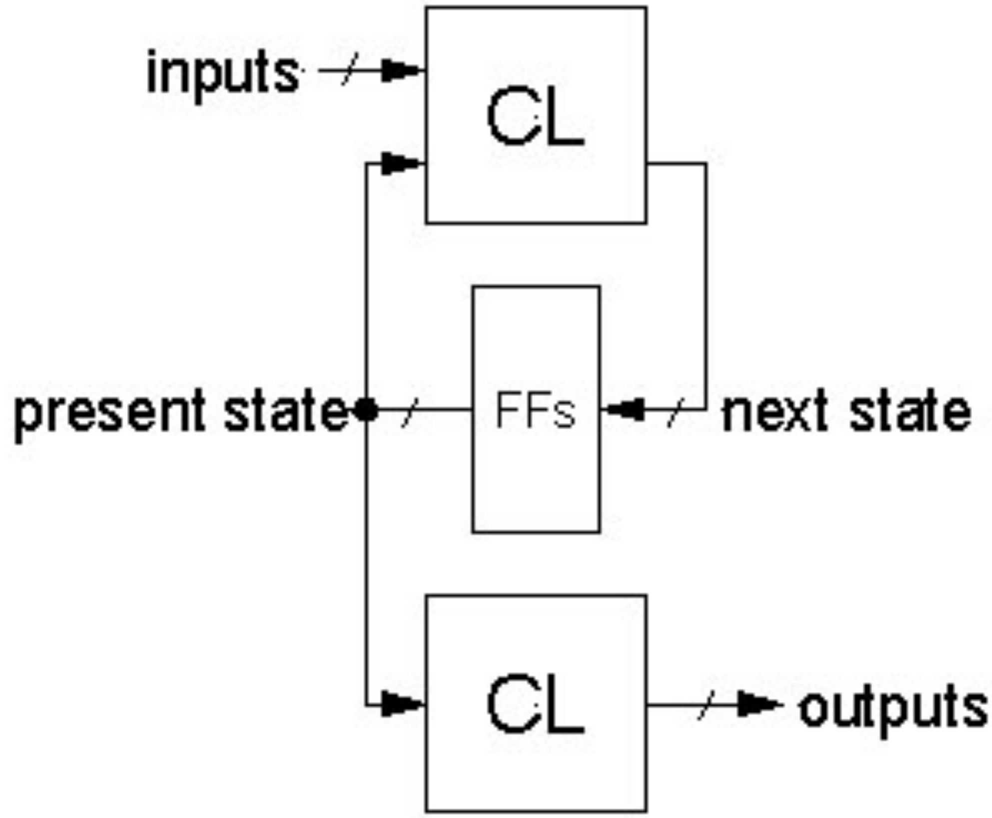


- Behavior is represented by ***State Transition Diagram***:
  - traverse one edge per clock cycle.



FSM Implementation





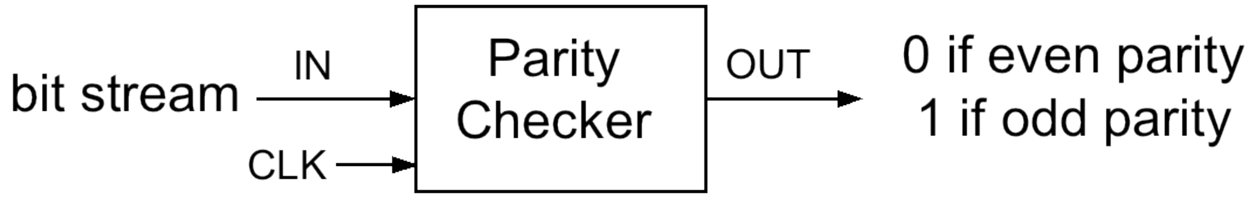
- Flip-flops form *state register*
- number of states ≤ $2^{\text{number of flip-flops}}$
- CL (combinational logic) calculates next state and output
- Remember: The FSM follows exactly one edge per cycle.

Later we will learn how to implement in Verilog. Now we learn how to design "by hand" to the gate level.

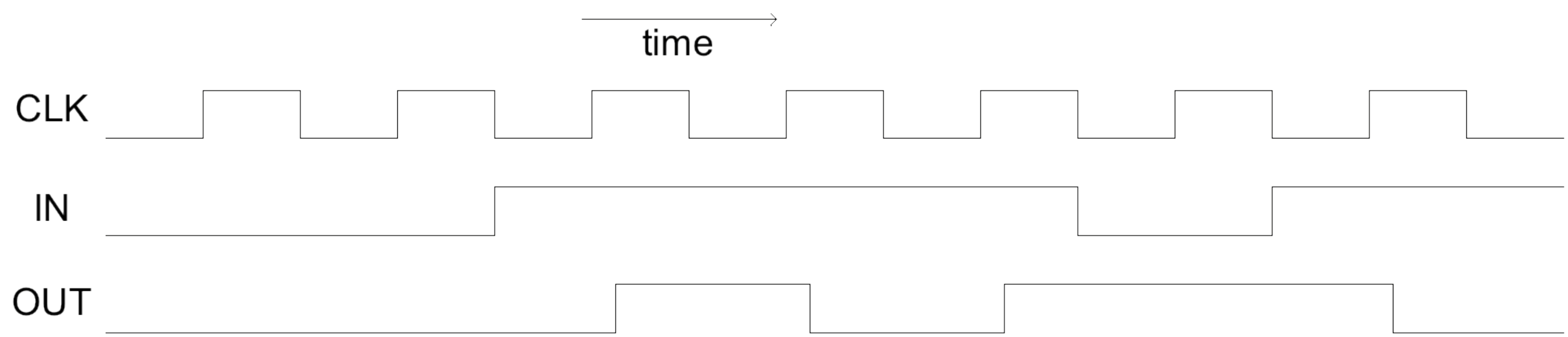
Parity Checker Example

A string of bits has "even parity" if the number of 1's in the string is even.

- Design a circuit that accepts a bit-serial stream of bits, and outputs a 0 if the parity thus far is even and outputs a 1 if odd:



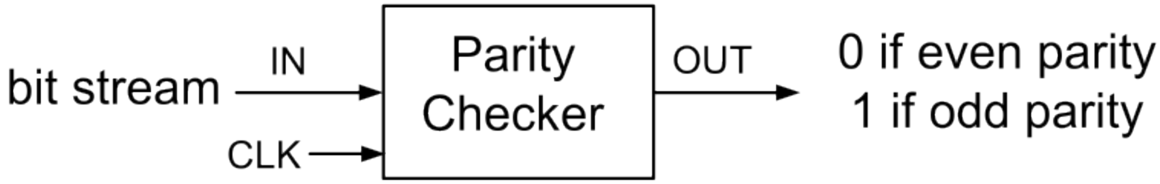
example:	0	0	1	1	1	0	1
	even	even	odd	even	odd	odd	even



Next we take this example through the "formal design process". But first, can you guess a circuit that performs this function?

Parity Counter

Formal Design Process (2)

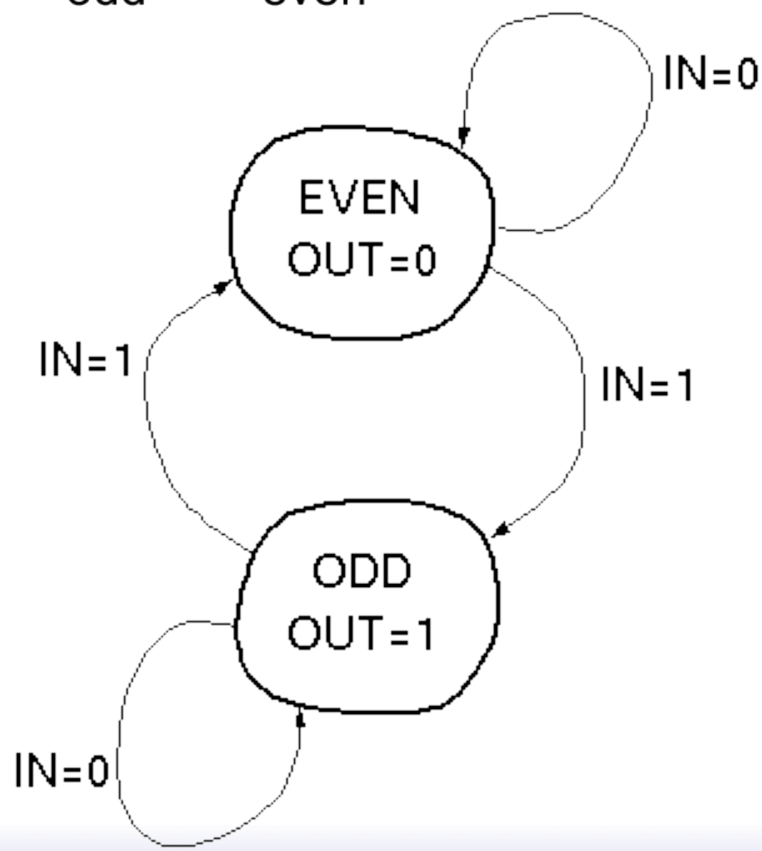


example:	0	0	1	1	1	0	1
	even	even	odd	even	odd	odd	even

time

"State Transition Diagram"

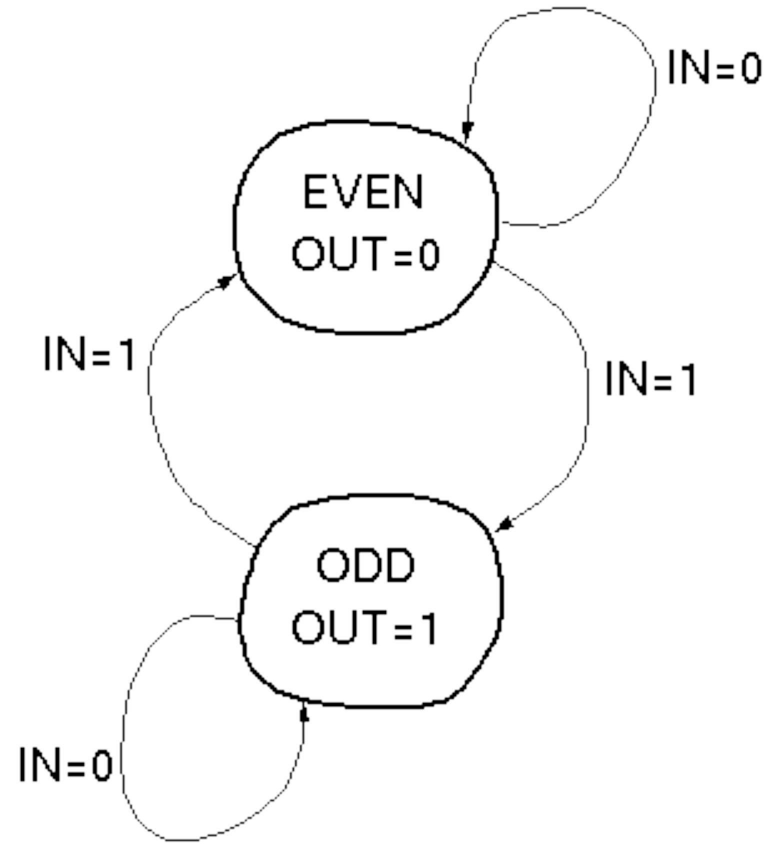
- circuit is in one of two "states".
- transition on each cycle with each new input, over exactly one arc (edge).
- Output depends on which state the circuit is in.



Formal Design Process (3,4)

State Transition Table:

present state	OUT	IN	next state
EVEN	0	0	EVEN
EVEN	0	1	ODD
ODD	1	0	ODD
ODD	1	1	EVEN



Invent a code to represent states:

Let 0 = EVEN state, 1 = ODD state

present state (ps)	OUT	IN	next state (ns)
0	0	0	0
0	0	1	1
1	1	0	1
1	1	1	0

Derive logic equations from table (how?):

OUT = PS

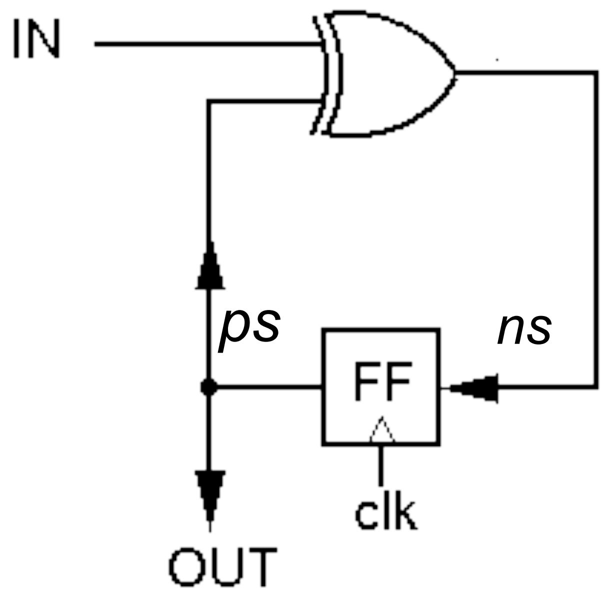
NS = PS xor IN

Formal Design Process (5,6)

Logic equations from table:

OUT = PS

NS = PS xor IN



- Circuit Diagram:
  - XOR gate for NS calculation
  - DFF to hold present state
  - no logic needed for output in this example.

Formal Design Process

Review of Design Steps:

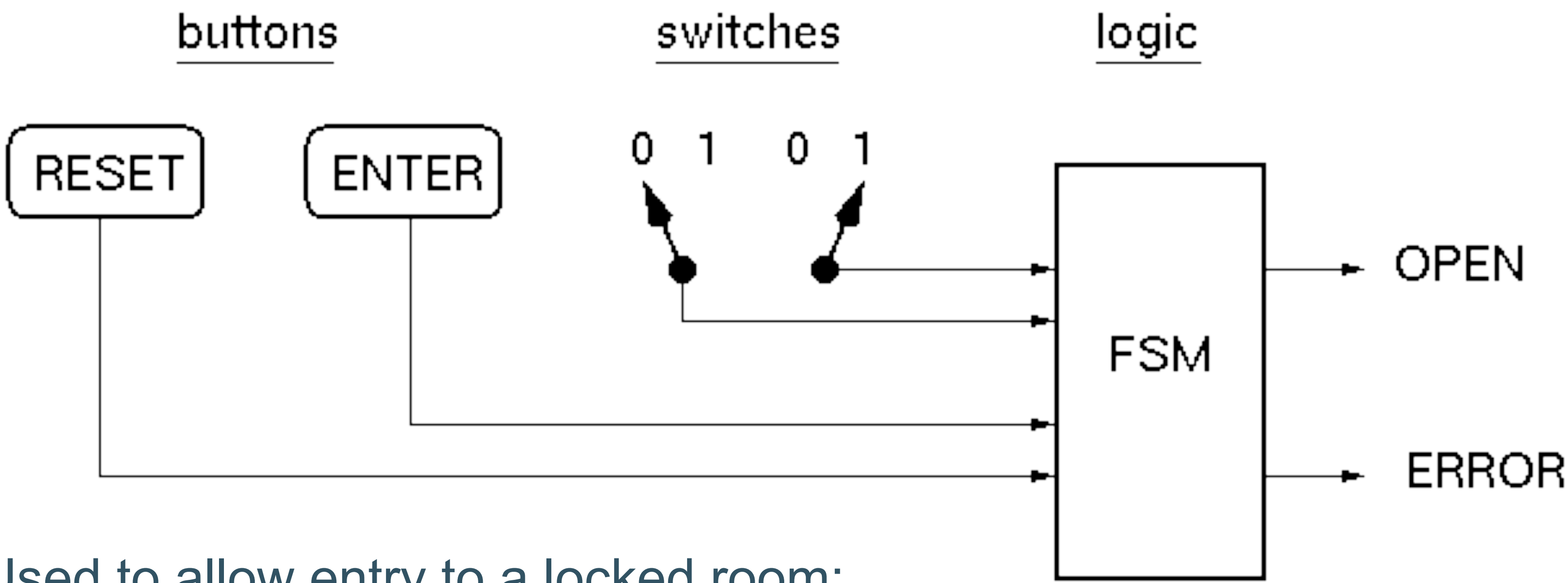
1. *Specify* ***circuit function*** *(English)*
2. *Draw* ***state transition diagram***
3. *Write down* ***symbolic state transition table***
4. *Write down* ***encoded state transition table***
5. *Derive* ***logic equations***
6. *Derive* ***circuit diagram***

    Register to hold state

    Combinational Logic for Next State and Outputs

FSM Design Example

Combination Lock Example



- Used to allow entry to a locked room:

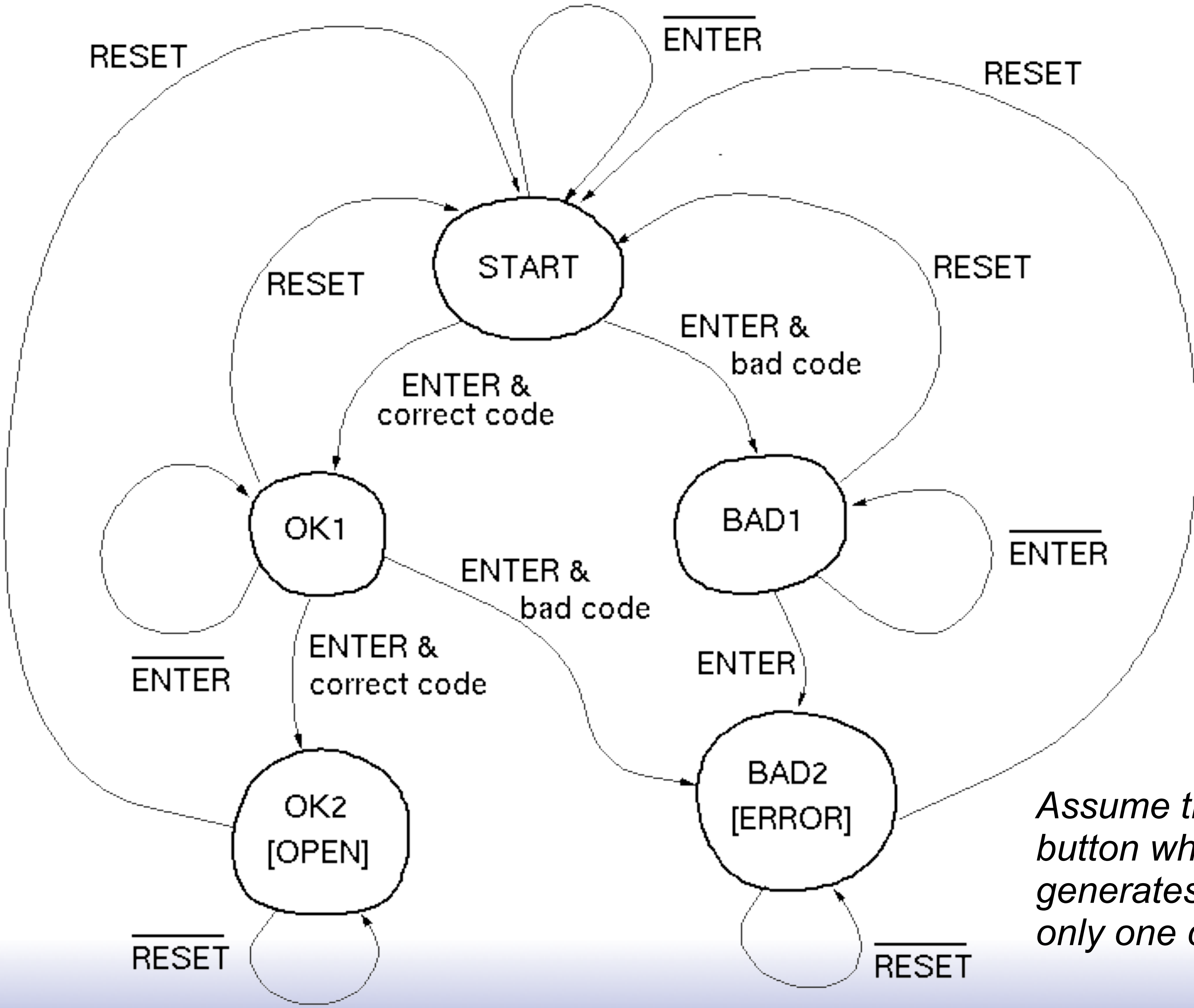
  2-bit serial combination. Example 01,11:

  1. Set switches to 01, press ENTER
  2. Set switches to 11, press ENTER
  3. OPEN is asserted (OPEN=1).

     If wrong code, ERROR is asserted (after second combo word entry).

     Press Reset at anytime to try again.

Combinational Lock STD

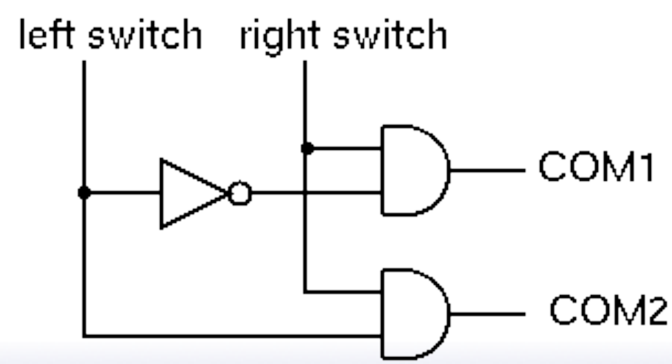


Assume the ENTER button when pressed generates a pulse for only one clock cycle.

Symbolic State Transition Table

RESET	ENTER	COM1	COM2	Preset State	Next State	OPEN	ERROR
0	0	*	*	START	START	0	0
0	1	0	*	START	BAD1	0	0
0	1	1	*	START	OK1	0	0
0	0	*	*	OK1	OK1	0	0
0	1	*	0	OK1	BAD2	0	0
0	1	*	1	OK1	OK2	0	0
0	*	*	*	OK2	OK2	1	0
0	0	*	*	BAD1	BAD1	0	0
0	1	*	*	BAD1	BAD2	0	0
0	*	*	*	BAD2	BAD2	0	1
1	*	*	*	*	START	0	0

Decoder logic for checking combination (01,11):



Encoded ST Table

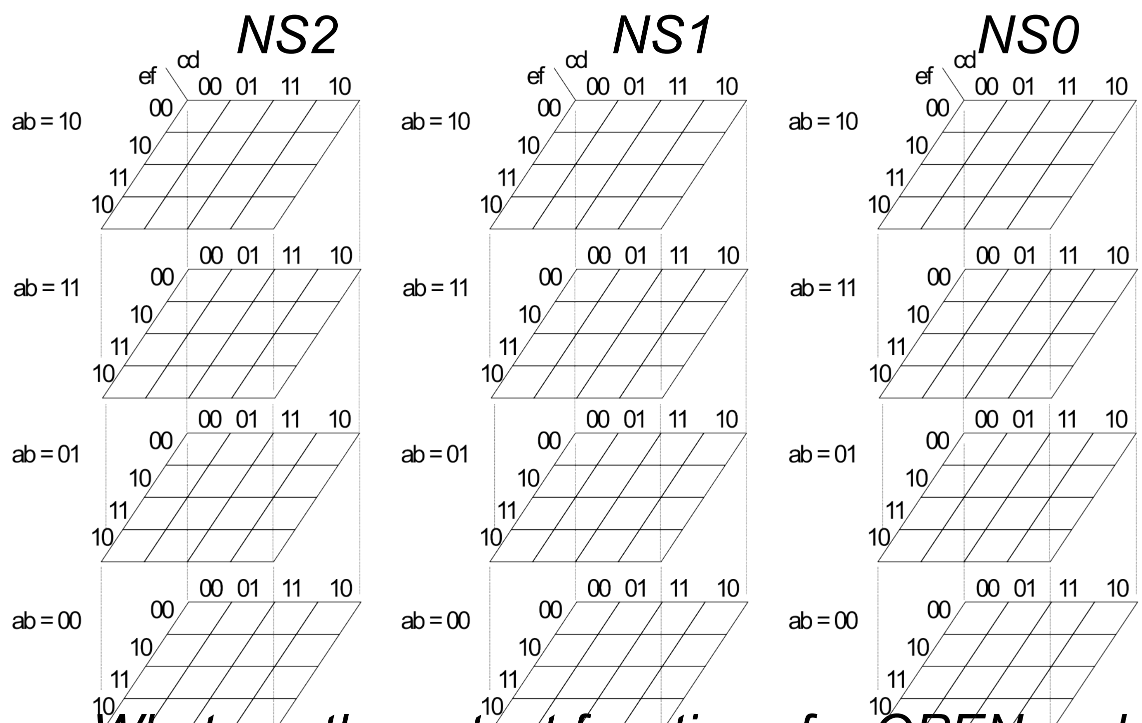
ENTER	COM1	COM2	PS2	PS1	PS0	NS2	NS1	NS0
0	0	0	0	0	0	0	0	0
0	0	1	0	0	0	0	0	0
0	1	0	0	0	0	0	0	0
0	1	1	0	0	0	0	0	0
1	0	0	0	0	0	1	0	0
1	0	1	0	0	0	1	0	0
1	1	0	0	0	0	0	0	1
1	1	1	0	0	0	0	0	1
0	0	0	0	0	1	0	0	1
0	0	1	0	0	1	0	0	1
0	1	0	0	0	1	0	0	1
0	1	1	0	0	1	0	0	1
1	0	0	0	0	1	1	0	1
1	1	0	0	0	1	1	0	1
1	0	1	0	0	1	0	1	1
1	1	1	0	0	1	0	1	1
0	0	0	0	1	1	0	1	1
0	0	1	0	1	1	0	1	1
0	1	0	0	1	1	0	1	1
0	1	1	0	1	1	0	1	1
1	0	0	0	1	1	0	1	1
1	0	1	0	1	1	0	1	1
1	1	0	0	1	1	0	1	1
1	1	1	0	1	1	0	1	1
0	0	0	1	0	0	1	0	0
0	0	1	1	0	0	1	0	0
0	1	0	1	0	0	1	0	0
0	1	1	1	0	0	1	0	0
1	0	0	1	0	0	1	0	1
1	0	1	1	0	0	1	0	1
1	1	0	1	0	0	1	0	1
1	1	1	1	0	0	1	0	1
0	0	0	1	0	1	1	0	1
0	0	1	1	0	1	1	0	1
0	1	0	1	0	1	1	0	1
0	1	1	1	0	1	1	0	1
1	0	0	1	0	1	1	0	1
1	0	1	1	0	1	1	0	1
1	1	0	1	0	1	1	0	1
1	1	1	1	0	1	1	0	1

- *Assign states:*

START=000, OK1=001, OK2=011

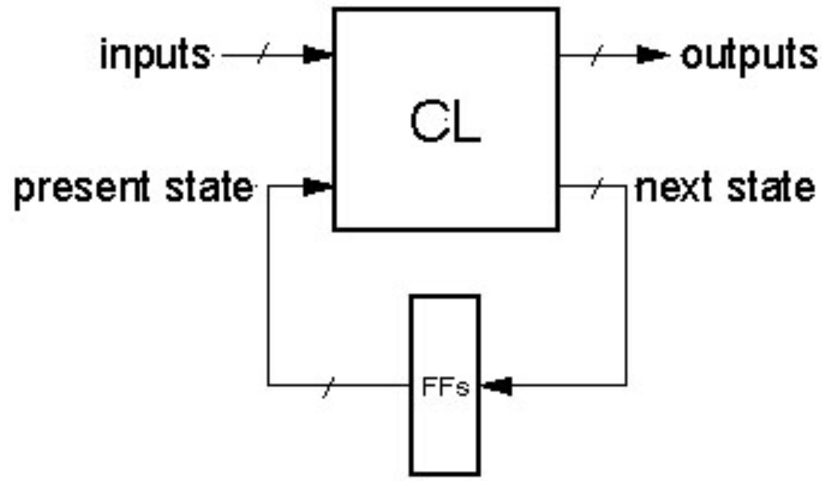
BAD1=100, BAD2=101

- *Omit reset. Assume that primitive flip-flops has reset input.*
- *Rows not shown have don't cares in output. Correspond to invalid PS values.*



- *What are the output functions for OPEN and ERROR?*

State Encoding



- In general:

  # of possible FSM states = $2^{\text{\# of Flip-flops}}$

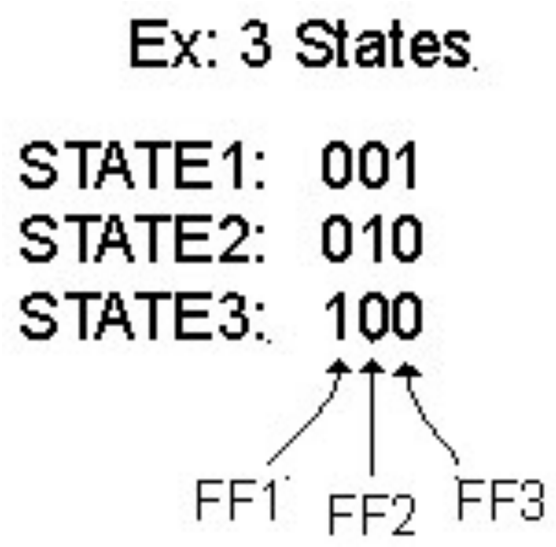
  Example:

  state1 = 01, state2 = 11, state3 = 10, state4 = 00

- However, often more than $\log_2$(# of states) FFs are used, to simplify logic at the cost of more FFs.
- Extreme example is one-hot state encoding.

State Encoding

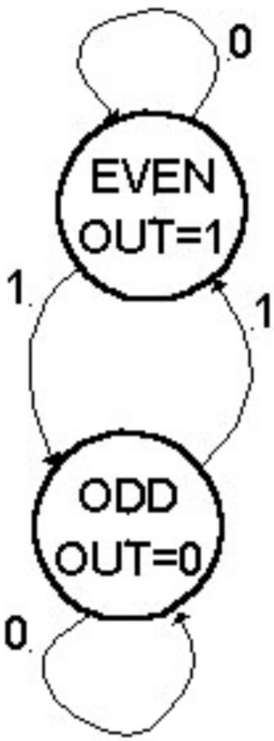
- **One-hot encoding of states.**
- One FF per state.

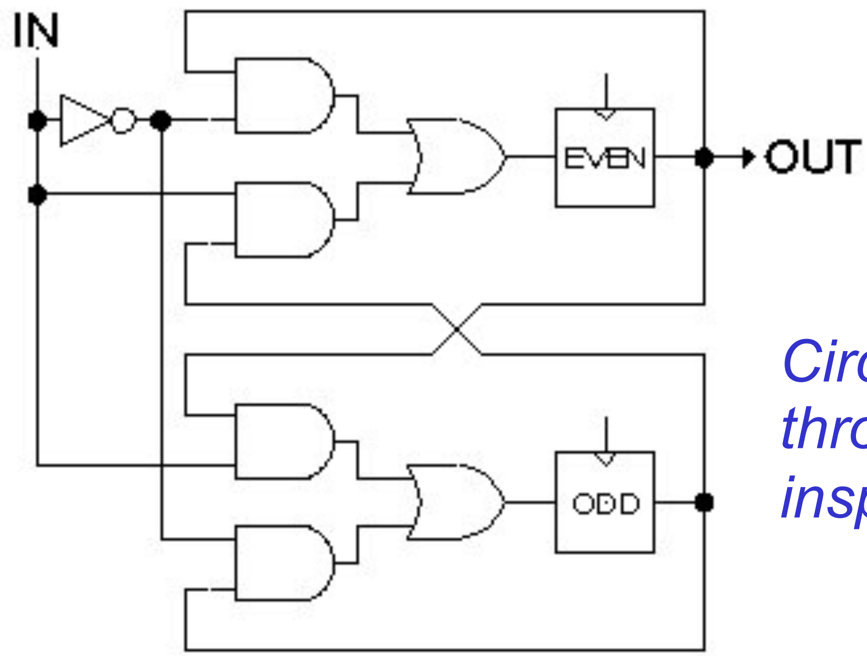


- Why one-hot encoding?
  - Simple design procedure.
    - Circuit matches state transition diagram (example next page).
  - Often can lead to simpler and faster “next state” and output logic.
- Why not do this?
  - Can be costly in terms of Flip-flops for FSMs with large number of states.
- FPGAs are “Flip-flop rich”, therefore one-hot state machine encoding is often a good approach.

One-hot encoded FSM

- Even Parity Checker Circuit:

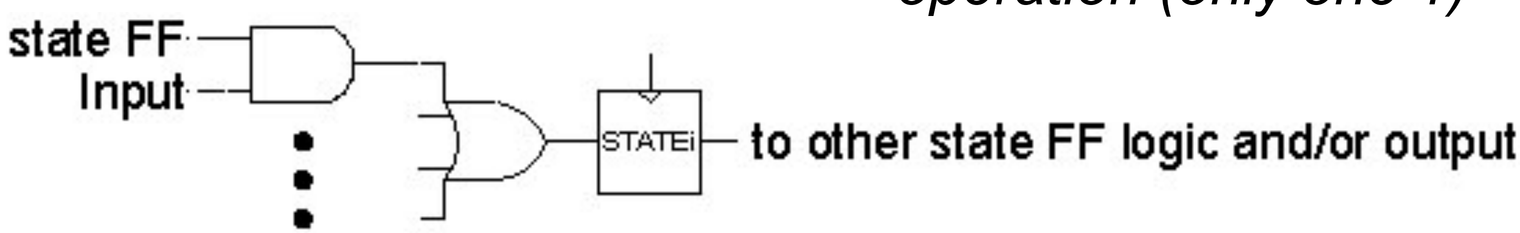




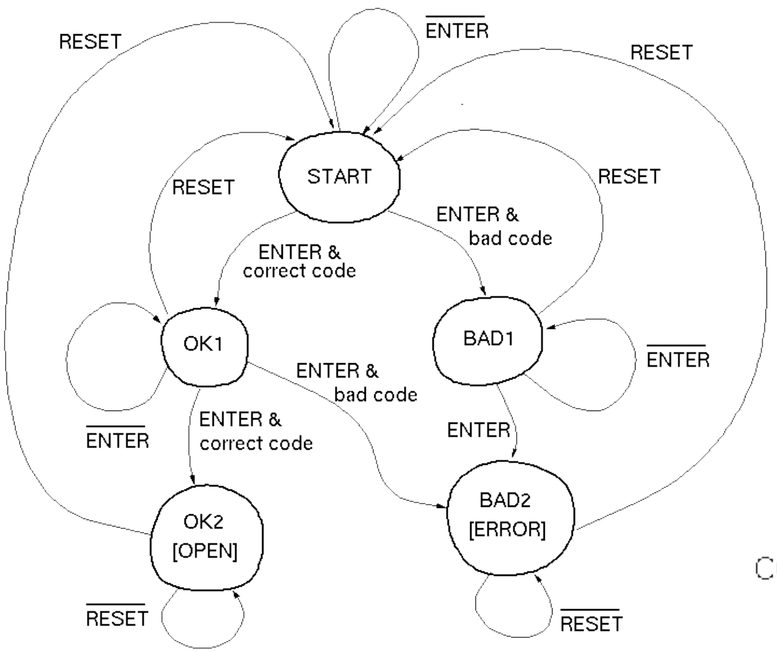
Circuit generated through direct inspection of the STD.

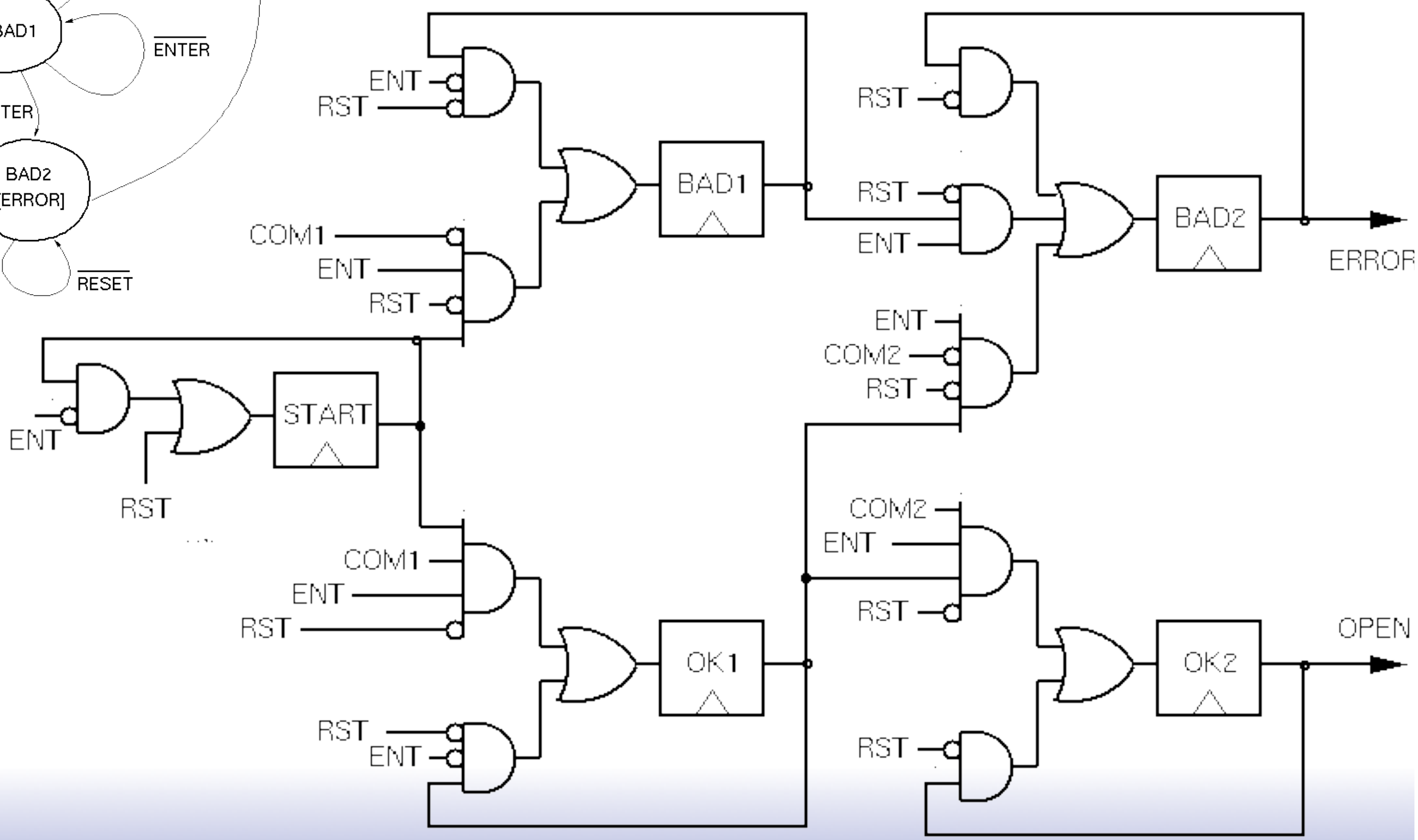
- In General:

- *FFs must be initialized for correct operation (only one 1)*



One-hot encoded combination lock

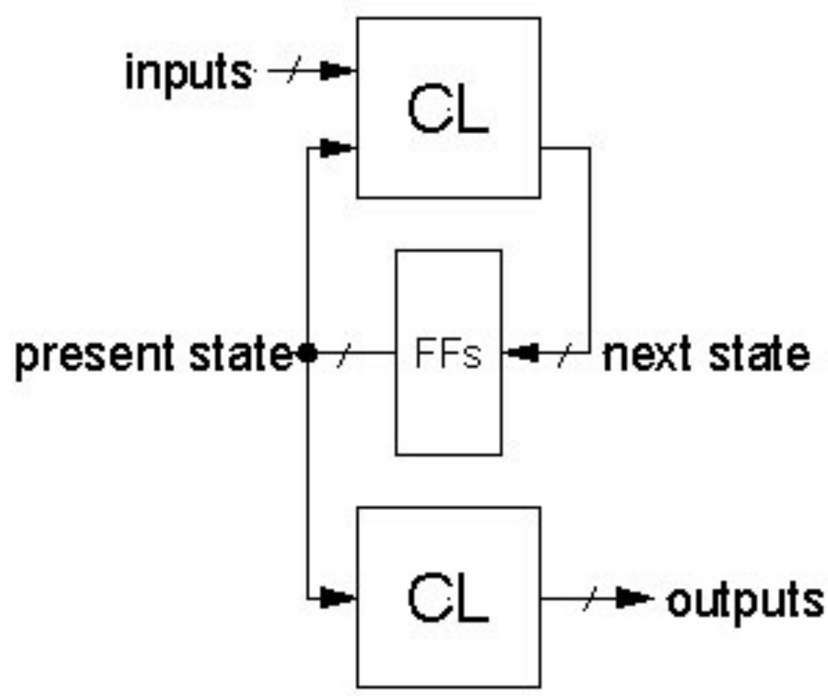




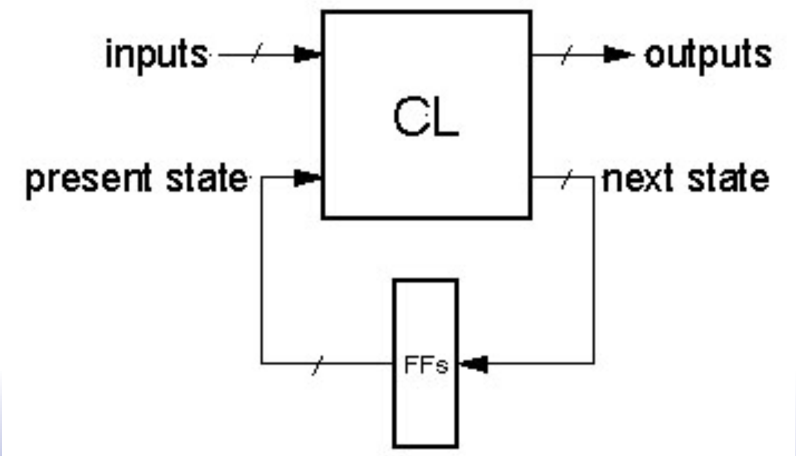
Moore Versus Mealy Machines

FSM Implementation Notes

- All examples so far generate output based only on the present state, commonly called a “Moore Machine”:



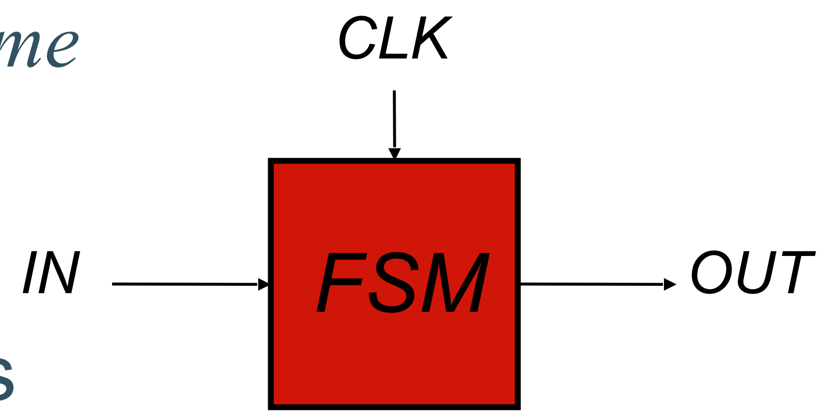
- If output functions include both present state and input then called a “*Mealy Machine*”:



Finite State Machines

- **Example: Edge Detector**

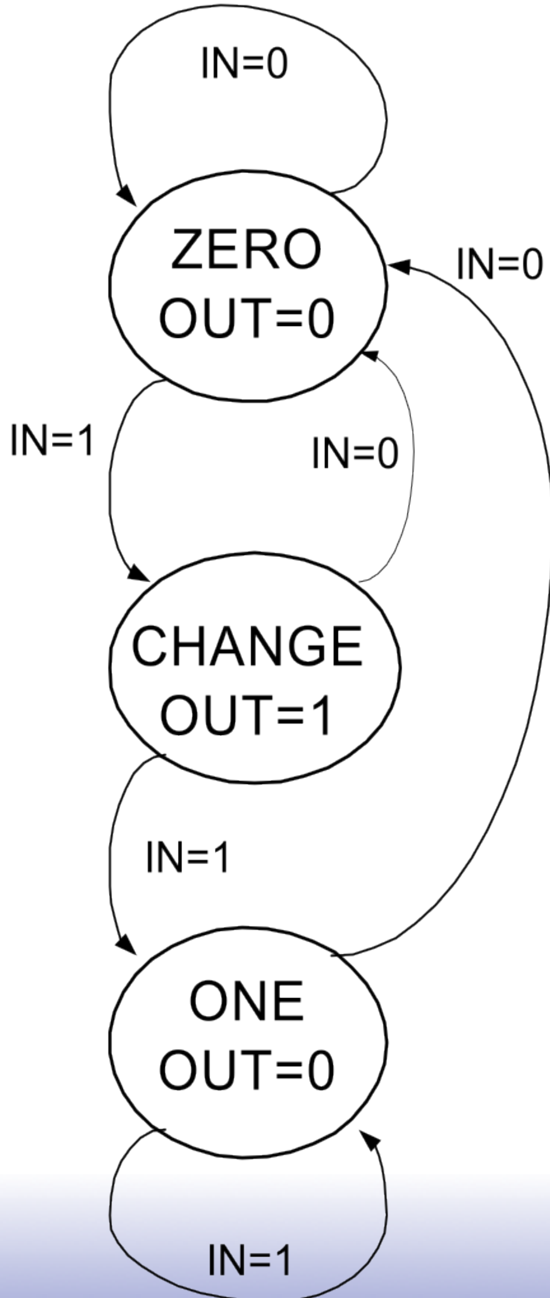
  Bit are received one at a time (one per cycle),
  such as: 000111010 → *time*



Design a circuit that asserts its output for one cycle when the input bit stream changes from 0 to 1.

We'll try two different solutions.

State Transition Diagram Solution A



	IN	PS	NS	OUT
ZERO	0	00	00	0
	1	00	01	0
CHANGE	0	01	00	1
	1	01	11	1
ONE	0	11	00	0
	1	11	11	0

Solution A, circuit derivation

	IN	PS	NS	OUT
ZERO	0	00	00	0
	1	00	01	0
CHANGE	0	01	00	1
	1	01	11	1
ONE	0	11	00	0
	1	11	11	0

IN \ PS	00	01	11	10
0	0	0	0	-
1	0	1	1	-

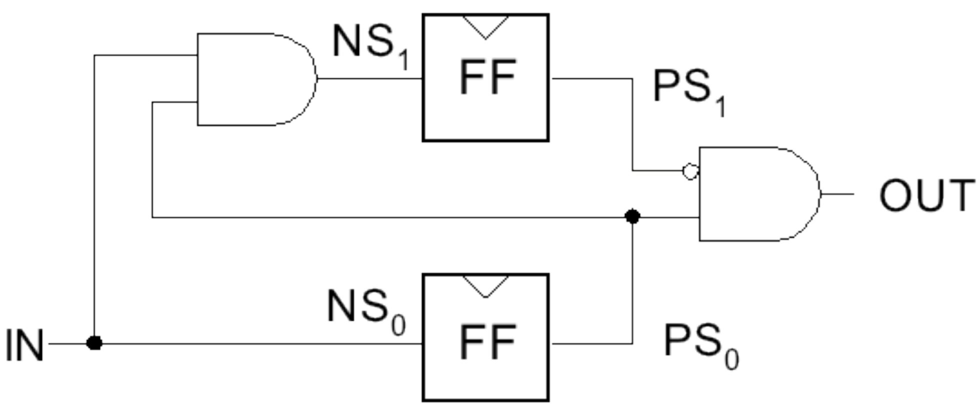
$NS_1 = IN\ PS_0$

IN \ PS	00	01	11	10
0	0	0	0	-
1	1	1	1	-

$NS_0 = IN$

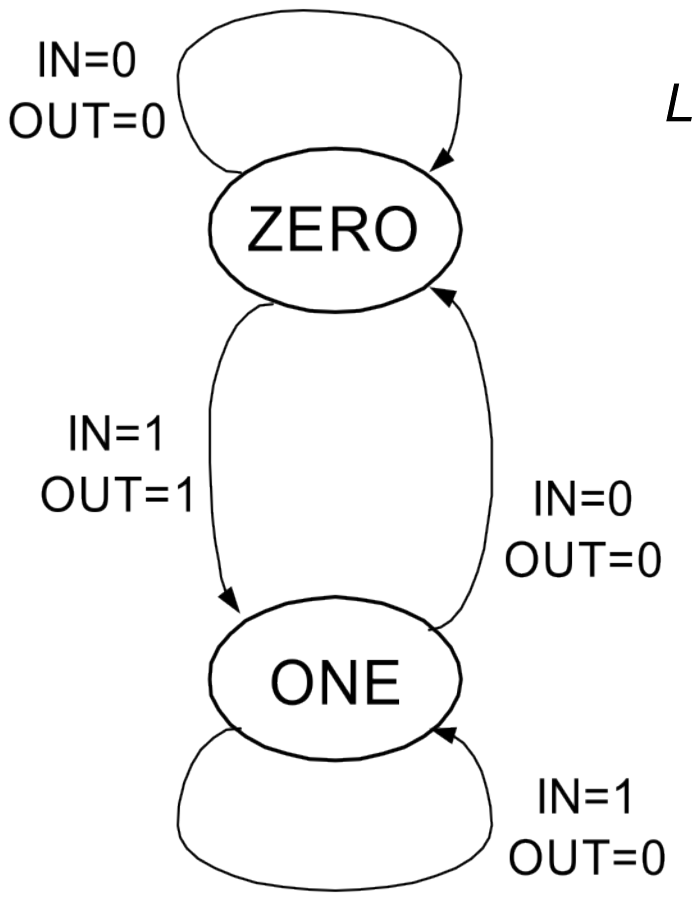
IN \ PS	00	01	11	10
0	0	1	0	-
1	0	1	0	-

$OUT = \overline{PS_1}\ PS_0$



Solution B

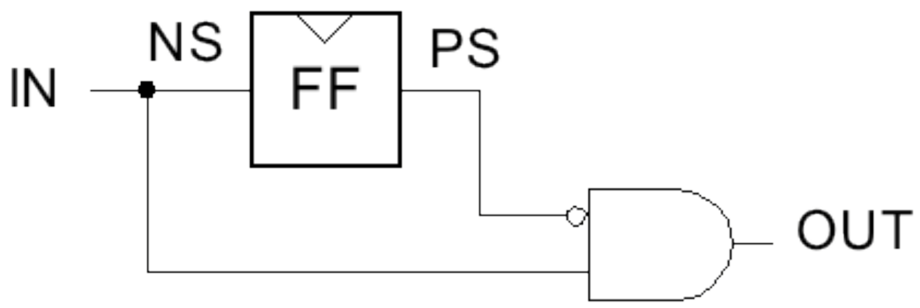
Output depends not only on PS but also on input, IN



Let ZERO=0, ONE=1

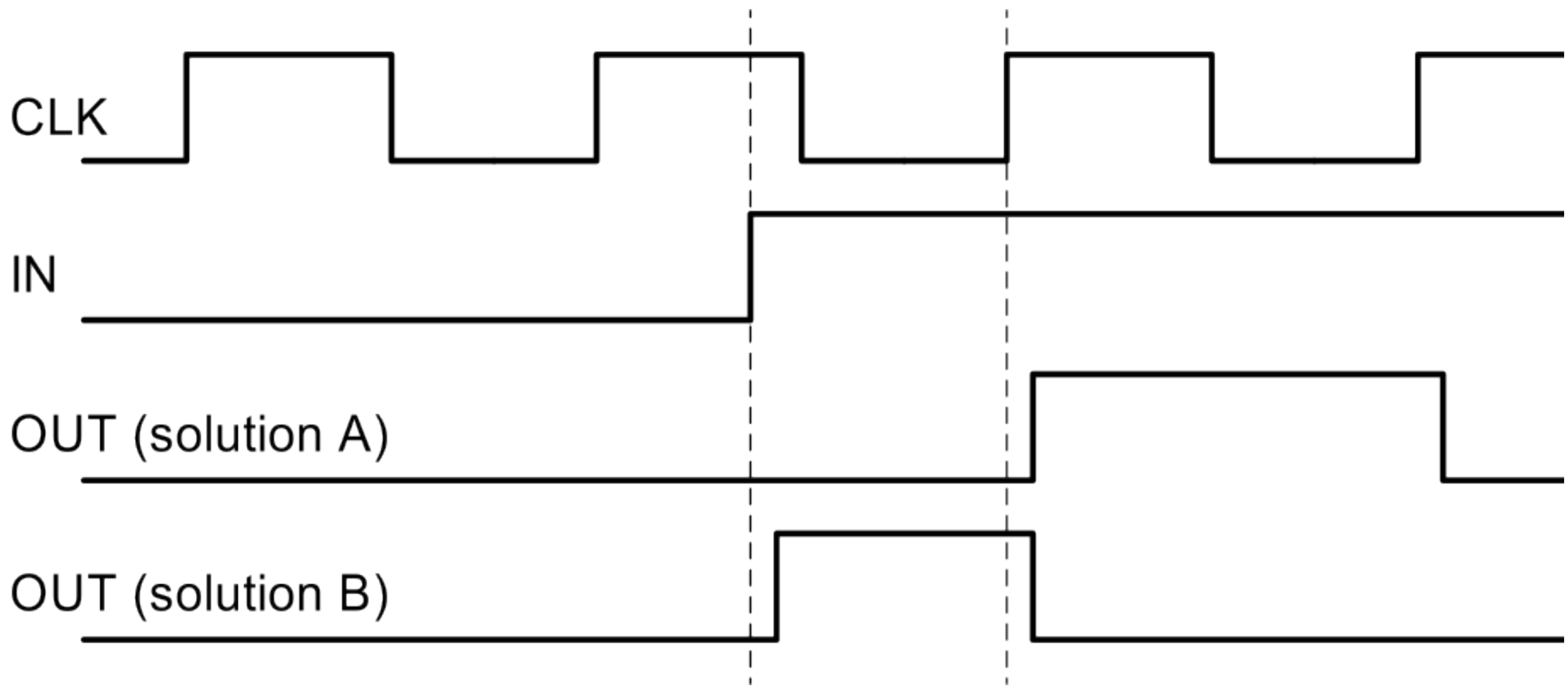
IN	*PS*	*NS*	*OUT*
0	0	0	0
0	1	0	0
1	0	1	1
1	1	1	0

NS = IN, OUT = IN PS'



What's the intuition about this solution?

Edge detector timing diagrams



- *Solution A: output follows the clock*
- *Solution B: output changes with input rising edge and is asynchronous wrt the clock.*

FSM Comparison

Solution A

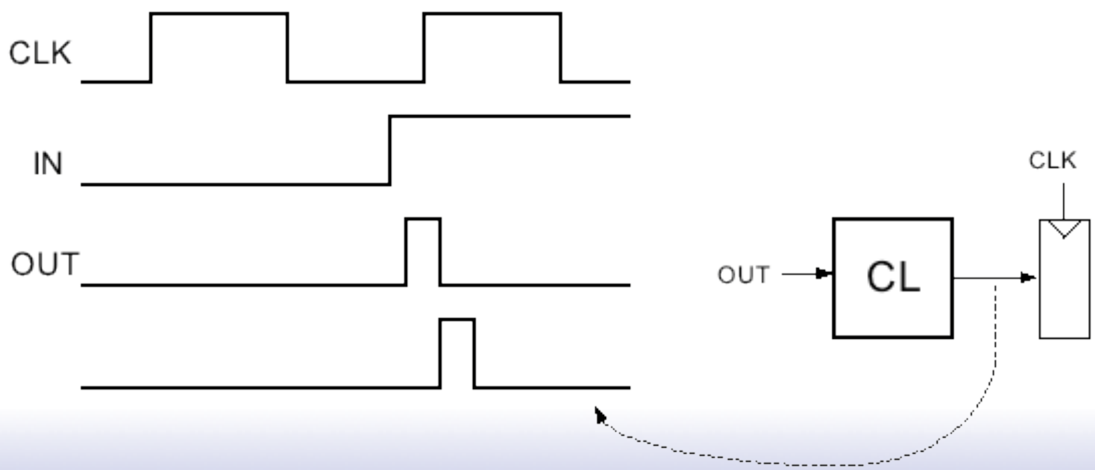
Moore Machine

- output function only of PS
- maybe <u>more</u> states (why?)
- synchronous outputs
  - Input glitches not send at output
  - one cycle "delay"
  - full cycle of stable output

Solution B

Mealy Machine

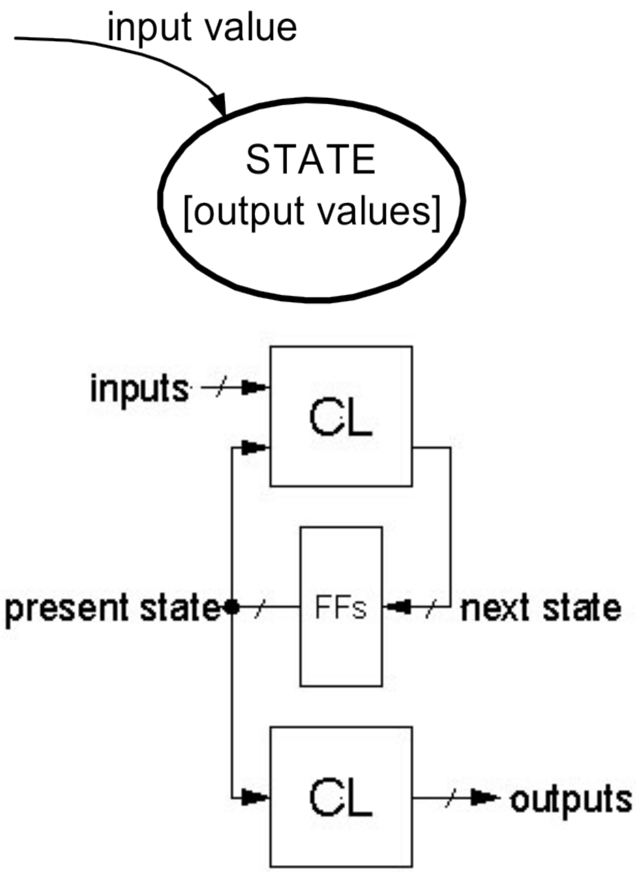
- *output function of both PS & input*
- *maybe fewer states*
- *asynchronous outputs*
  - *if input glitches, so does output*
  - *output immediately available*
  - *output may not be stable long enough to be useful (below):*



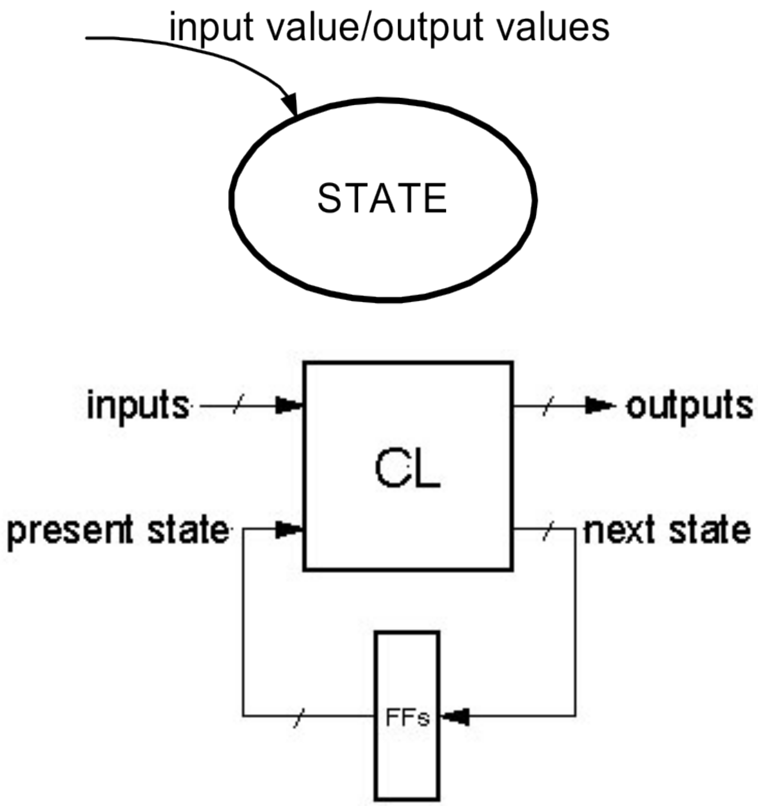
If output of Mealy FSM goes through combinational logic before being registered, the CL might delay the signal and it could be missed by the clock edge.

FSM Recap

Moore Machine



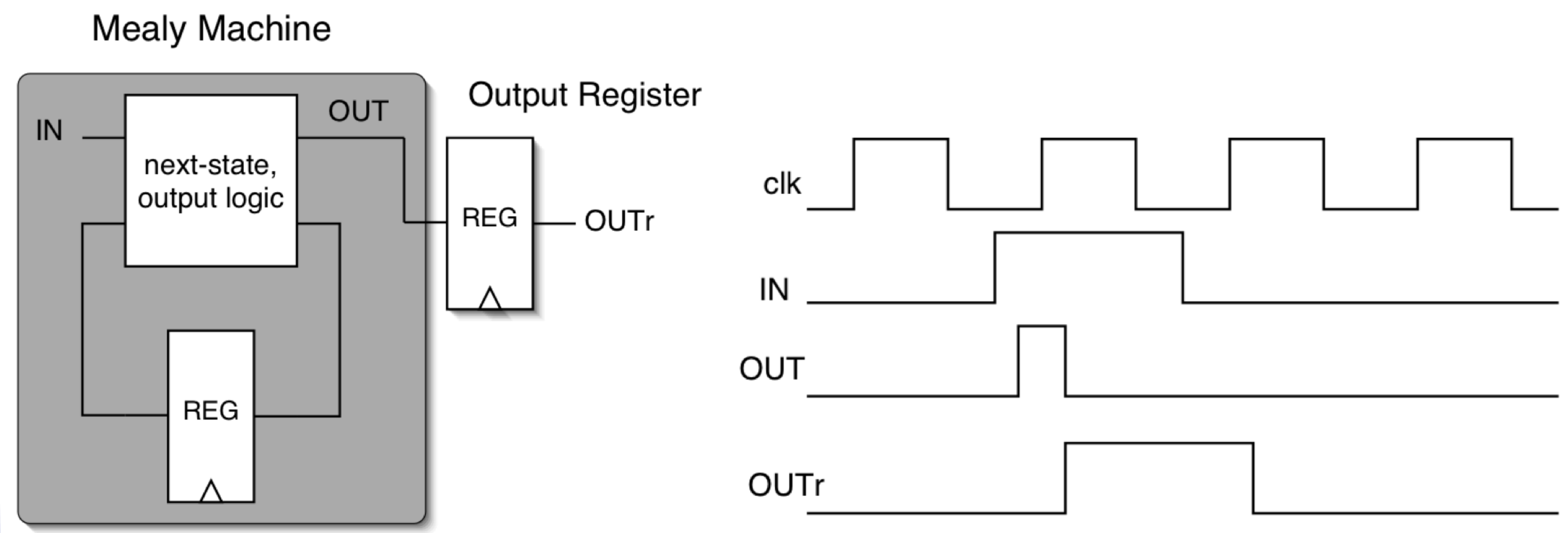
Mealy Machine



Both machine types allow one-hot implementations.

Final Notes on Moore versus Mealy

1. A given state machine *could* have *both* Moore and Mealy style outputs. Nothing wrong with this, but you need to be aware of the timing differences between the two types.
2. The output timing behavior of the Moore machine can be achieved in a Mealy machine by “registering” the Mealy output values:



FSMs in Verilog

General FSM Design Process with Verilog Implementation

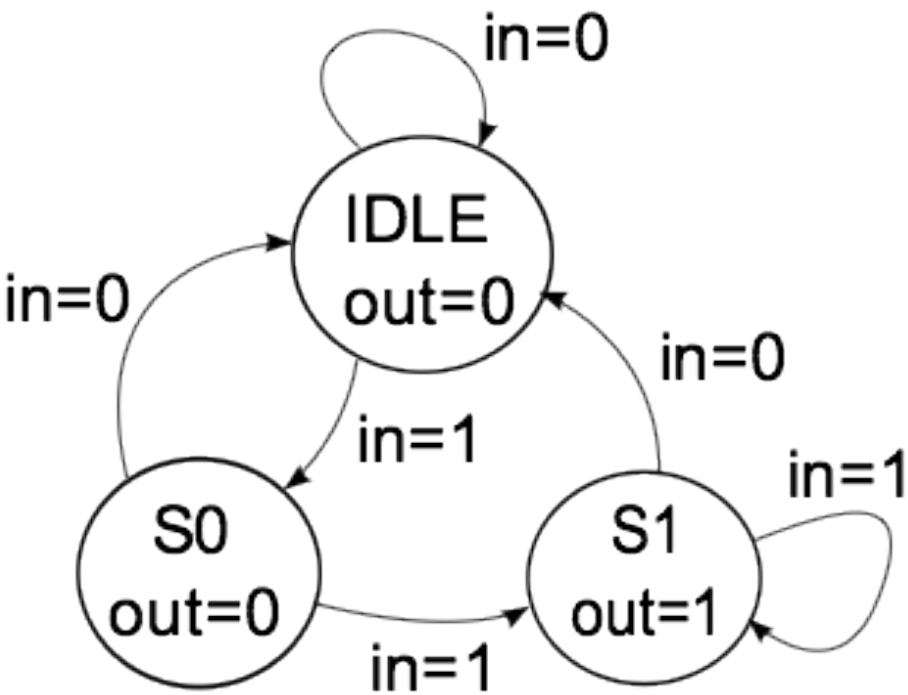
Design Steps:

1. *Specify **circuit function** (English)*
2. *Draw **state transition diagram***
3. *Write down **symbolic state transition table***
4. *Assign encodings (bit patterns) to symbolic states*
5. *Code as Verilog behavioral description*

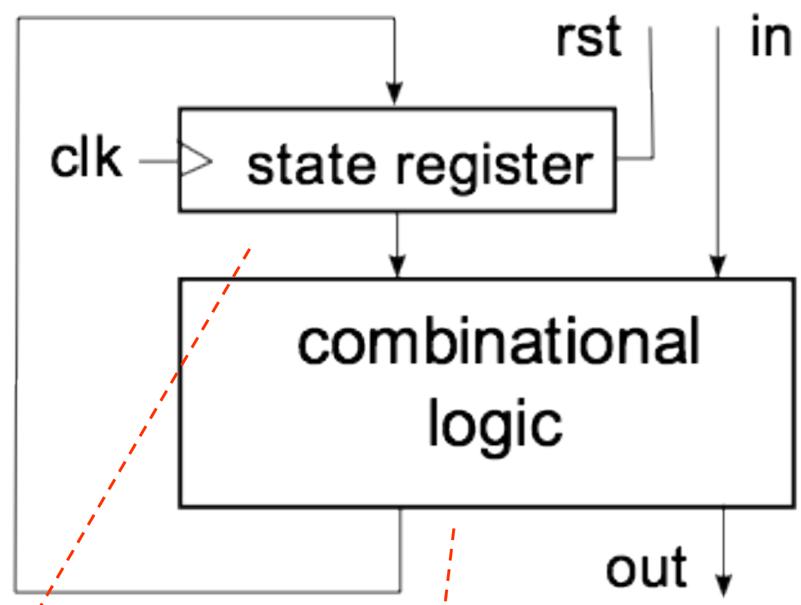
- ✓ *Use parameters to represent encoded states.*
- ✓ *Use separate always blocks for register assignment and CL logic block.*
- ✓ *Use case for CL block. Within each case section (state) assign all outputs and next state value based on inputs. Note: For Moore style machine make outputs dependent only on state not dependent on inputs.*

Finite State Machine in Verilog

State Transition Diagram



Implementation Circuit Diagram



Holds a symbol to keep track of which bubble the FSM is in.

CL functions to determine output value and next state based on input and current state.

out = f(in, current state)

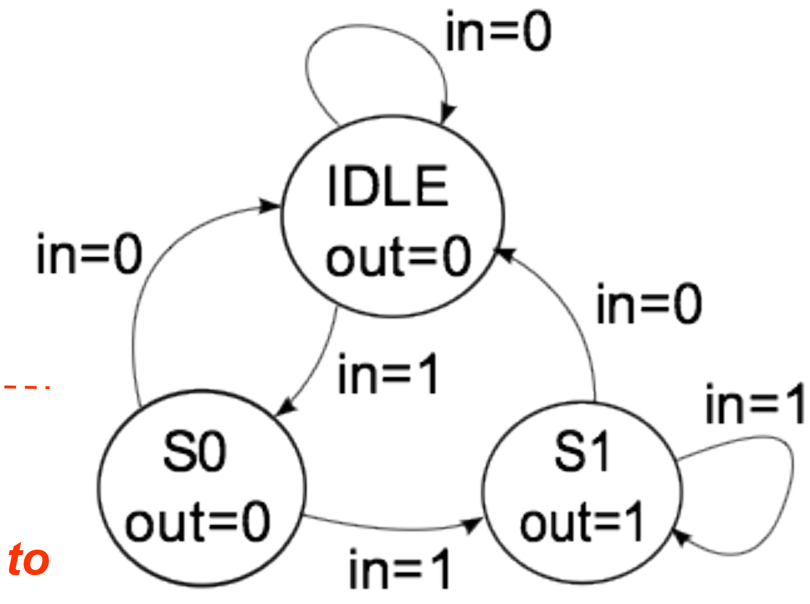
next state = f(in, current state)

Finite State Machines

```
module FSM1(clk, rst, in, out);
input clk, rst;
input in;
output out;
```

Must use reset to force to initial state.

reset not always shown in STD




```
// Defined state encoding:
parameter IDLE = 2'b00;
parameter S0 = 2'b01;
parameter S1 = 2'b10;
reg out;
reg [1:0] present_state, next_state;
```

Constants local to this module.

out not a register, but assigned in always block

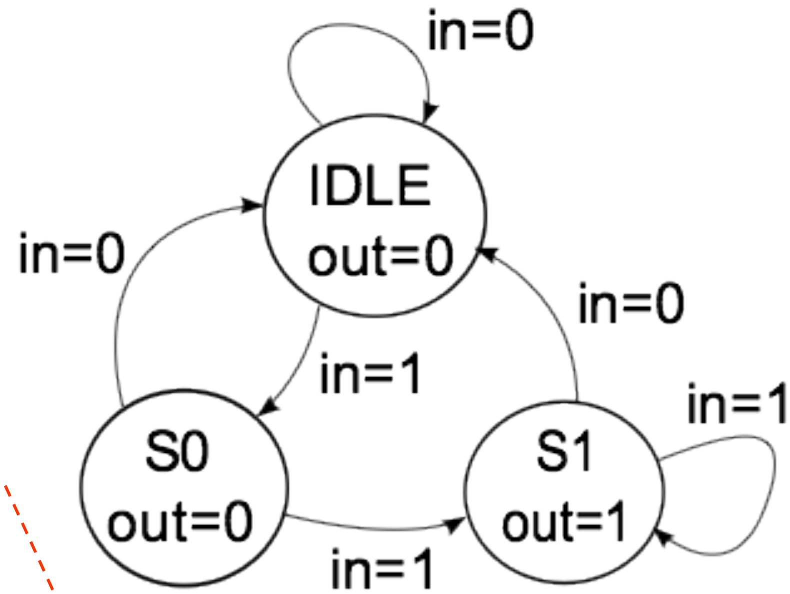
Combinational logic signals for transition.

THE register to hold the "state" of the FSM.

```
// always block for state register
always @(posedge clk)
      if (rst) present_state <= IDLE;
      else present_state <= next_state;
```

A separate always block should be used for combination logic part of FSM. Next state and output generation. (Always blocks in a design work in parallel.)

FSMs (cont.)




```
// always block for combinational logic portion
always @(present_state or in)
case (present_state)
// For each state def output and next
  IDLE    : begin
              out = 1'b0;
              if (in == 1'b1) next_state = S0;
              else next_state = IDLE;
            end
  S0      : begin
              out = 1'b0;
              if (in == 1'b1) next_state = S1;
              else next_state = IDLE;
            end
  S1      : begin
              out = 1'b1;
              if (in == 1'b1) next_state = S1;
              else next_state = IDLE;
          end
  default: begin
              next_state = IDLE;
              out = 1'b0;
            end
endcase
endmodule
```

Each state becomes a case clause.

For each state define:

Output value(s)

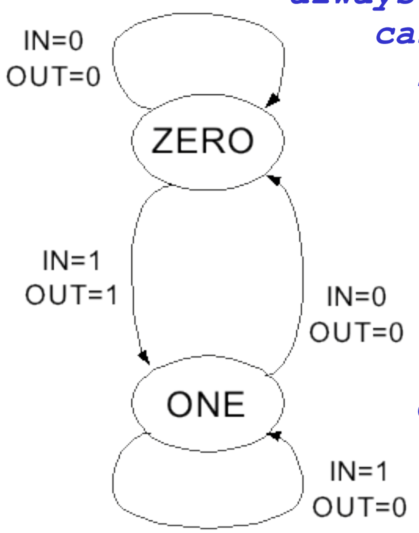
State transition

Use "default" to cover unassigned state. Usually unconditionally transition to reset state.

Mealy or Moore?

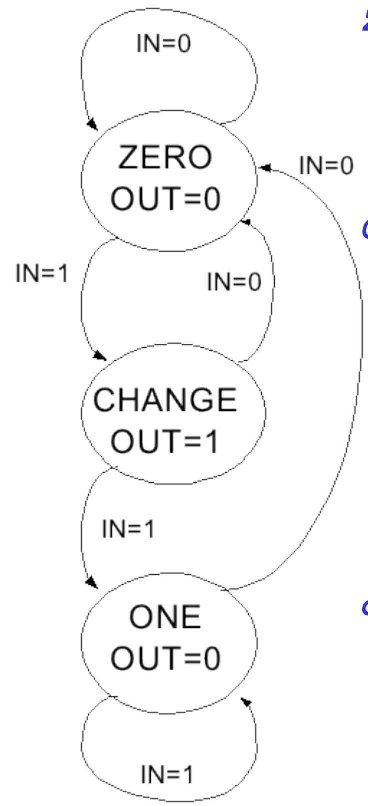
Edge Detector Example

Mealy Machine




```
always @(posedge clk)
        if (rst) ps <= ZERO;
        else ps <= ns;
always @(ps in)
    case (ps)
      ZERO: if (in) begin
              out = 1'b1;
              ns = ONE;
            end
            else begin
              out = 1'b0;
              ns = ZERO;
            end
      ONE: if (in) begin
             out = 1'b0;
             ns = ONE;
           end
           else begin
             out = 1'b0;
             ns = ZERO;
           end
      default: begin
               out = 1'bx;
               ns = default;
             end
```

Moore Machine




```
always @(posedge clk)
        if (rst) ps <= ZERO;
        else ps <= ns;
always @(ps in)
    case (ps)
      ZERO: begin
              out = 1'b0;
              if (in) ns = CHANGE;
               else ns = ZERO;
            end
      CHANGE: begin
               out = 1'b1;
               if (in) ns = ONE;
               else ns = ZERO;
             end
        ONE: begin
               out = 1'b0;
               if (in) ns = ONE;
               else ns = ZERO;
      default: begin
               out = 1'bx;
               ns = default;
             end
```

Procedural Assignments

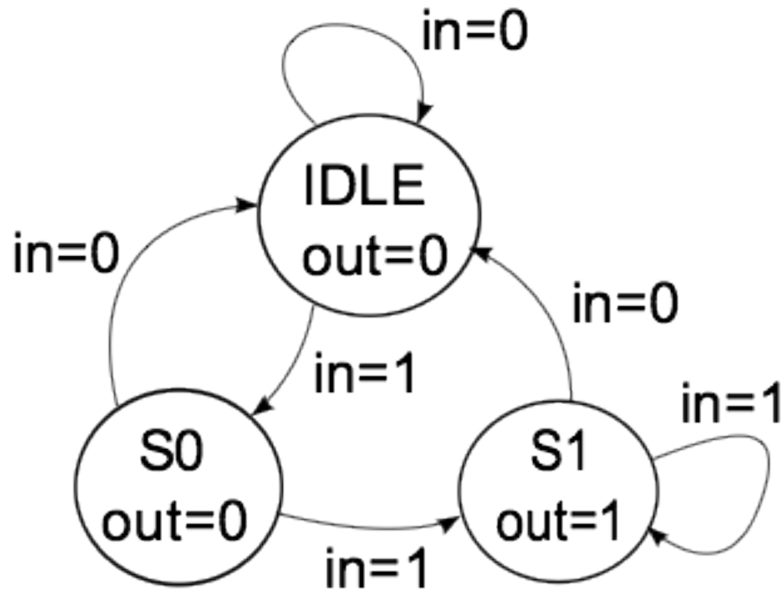
The sequential semantics of the blocking assignment allows variables to be multiply assigned within a single always block. Unexpected behavior can result from mixing these assignments in a single block. Standard rules:

i. Use blocking assignments to model combinational logic within an always block (“=”).
ii. Use non-blocking assignments to implement sequential logic (“<=”).
iii. Do not mix blocking and non-blocking assignments in the same always block.
iv. Do not make assignments to the same variable from more than one always block.

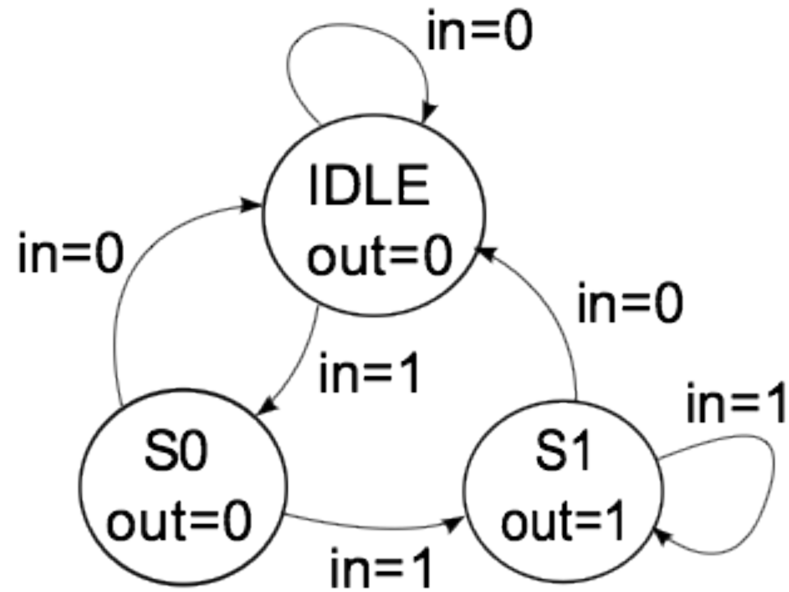
FSM CL block (original)

```
always @(present_state or in)
 case (present_state)
  IDLE    : begin
             out = 1'b0;
             if (in == 1'b1) next_state = S0;
             else next_state = IDLE;
            end
  S0      : begin
             out = 1'b0;
             if (in == 1'b1) next_state = S1;
             else next_state = IDLE;
            end
  S1      : begin
             out = 1'b1;
             if (in == 1'b1) next_state = S1;
             else next_state = IDLE;

              end
  default: begin
              next_state = IDLE;
              out = 1'b0;
             end
 endcase
endmodule
```



FSM CL block rewritten




```
always @*
  begin
   next_state = IDLE;
   out = 1'b0;
   case (state)
    IDLE    : if (in == 1'b1) next_state = S0;
    S0      : if (in == 1'b1) next_state = S1;
    S1      : begin
               out = 1'b1;
               if (in == 1'b1) next_state = S1;
              end
    default: ;
   endcase
  end
Endmodule
```

** for sensitivity list*

Normal values: used unless specified below.

Within case only need to specify exceptions to the normal values.

Note: The use of "blocking assignments" allow signal values to be "rewritten", simplifying the specification.

Some final warnings

Combinational logic always blocks

Make sure all signals assigned in a combinational always block are explicitly assigned values every time that the always block executes. Otherwise latches will be generated to hold the last value for the signals not assigned values.

Sel case value 2'd2 omitted.

Out is not updated when select line has 2'd2.

Latch is added by tool to hold the last value of out under this condition.

Similar problem with if-else statements.

```
module mux4to1 (out, a, b, c, d, sel);
output out;
input a, b, c, d;
input [1:0] sel;
reg out;
always @(sel or a or b or c or d)
begin
  case (sel)
    2'd0: out = a;
    2'd1: out = b;
    2'd3: out = d;
  endcase
end
endmodule
```

Combinational logic always blocks (cont.)

To avoid synthesizing a latch in this case, add the missing select line:

```
2'd2: out = c;
```

Or, in general, use the "default" case:

```
default:  out = foo;
```

If you don't care about the assignment in a case (for instance you know that it will never come up) then you can assign the value "x" to the variable. Example:

```
default:  out = 1'bx;
```

The x is treated as a "don't care" for synthesis and will simplify the logic.

Be careful when assigning x (don't care). If this case were to come up, then the synthesized circuit and simulation **may differ**.

Incomplete Triggers

Leaving out an input trigger usually results in latch generation for the missing trigger.

```
module and_gate (out, in1, in2);
  input     in1, in2;
  output        out;
  reg           out;

  always @(in1) begin
     out = in1 & in2;
  end

endmodule
```

in2 not in always sensitivity list.

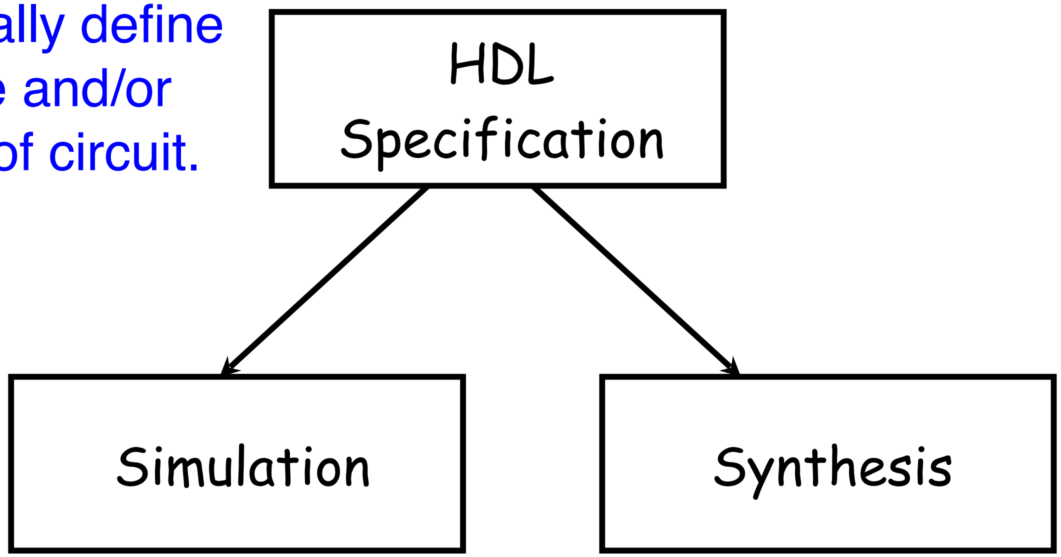
A latched version of in2 is synthesized and used as input to the and-gate, so that the and-gate output is not always sensitive to in2.

Easy way to avoid incomplete triggers for combinational logic is with: **`always @*`**

Intro to Logic Synthesis

EECS151/251A Design Methodology

Hierarchically define structure and/or behavior of circuit.



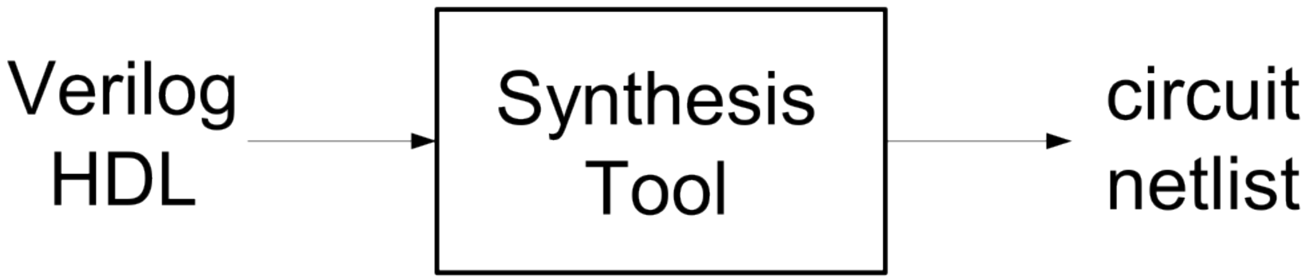
Functional verification.

Maps specification to resources of implementation platform (FPGA or ASIC).

Note: This in not the entire story. Other tools are often used analyze HDL specifications and synthesis results. More on this later.

Logic Synthesis

- Verilog and VHDL started out as simulation languages, but quickly people wrote programs to automatically convert Verilog code into low-level circuit descriptions (netlists).



- Synthesis converts Verilog (or other HDL) descriptions to implementation technology specific primitives:
  - For FPGAs: LUTs, flip-flops, and RAM blocks
  - For ASICs: standard cell gate and flip-flop libraries, and memory blocks.

Why Logic Synthesis?

1. Automatically manages many details of the design process:
   ⇒ Fewer bugs
   ⇒ Improved productivity
2. Abstracts the design data (HDL description) from any particular implementation technology.
   - Designs can be re-synthesized targeting different chip technologies. Ex: first implement in FPGA then later in ASIC.
3. In some cases, leads to a more optimal design than could be achieved by manual means (ex: logic optimization)

Why Not Logic Synthesis?

1. *May lead to non-optimal designs in some cases.*
2. *Often less transparent than desired: Good performance requires basically modeling the compiler in your head…*

`foo.v`

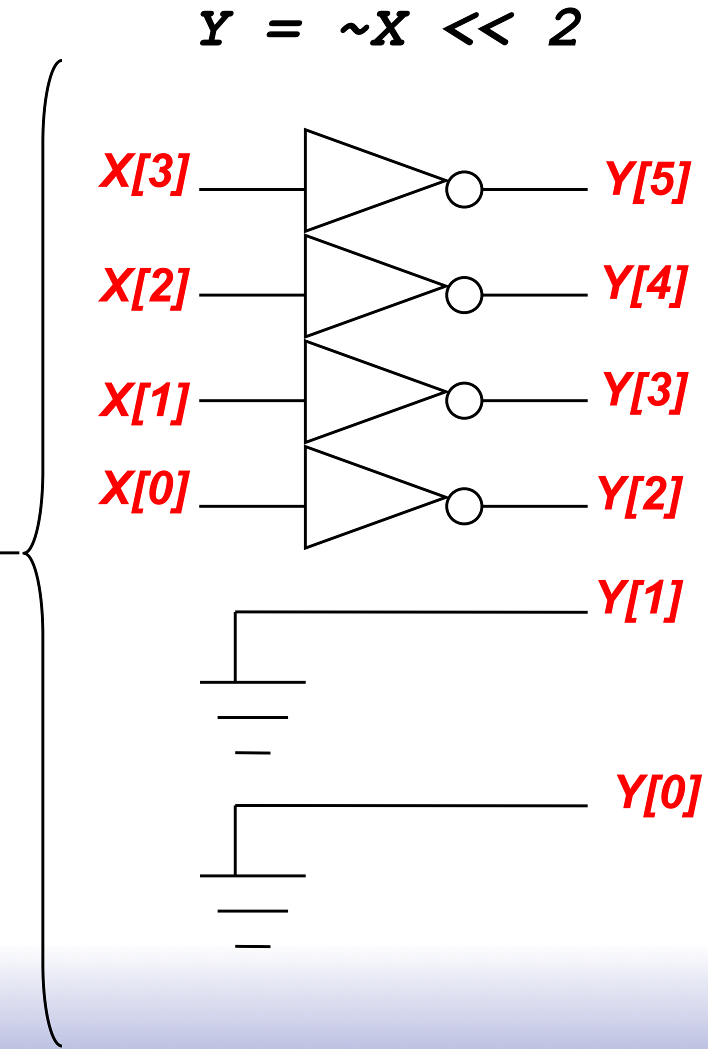
Main Logic Synthesis Steps

Parsing and Syntax Check	Load in HDL file, run macro preprocessor for \`define, \`include, etc..
Design Elaboration	Compute parameter expressions, process generates, create instances, connect ports.
Inference and Library Substitution	Recognize and insert special blocks (memory, flip-flops, arithmetic structures, ...)
Logic Expansion	Expand combinational logic to primitive Boolean representation.
Logic Optimization	Apply Boolean algebra and heuristics to simplify and optimize under constraints.
Map, Place & Route	CL and state elements to LUTs (FPGA) or Technology Library (ASCI) , assign physical locations, route connections.

`foo.ncd, foo.gates`

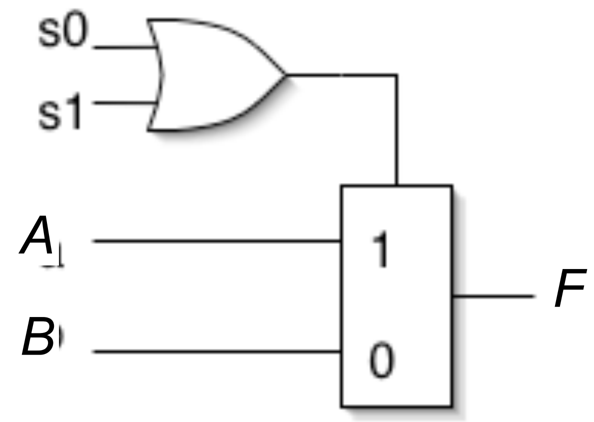
Operators and Synthesis

- Logical operators map into primitive logic gates
- Arithmetic operators map into adders, subtractors, …
  - Unsigned 2s complement
  - Model carry: target is one-bit wider that source
  - Watch out for *, %, and /
- Relational operators generate comparators
- Shifts by constant amount are just wire connections
  - No logic involved
- Variable shift amounts, a whole different story --- shifter
- Conditional expression generates logic or MUX



Simple Synthesis Example

```
module foo (A, B, s0, s1, F);
  input [3:0] A;
  input [3:0] B;
  input s0,s1;
  output [3:0] F;
  reg F;
  always @ (*)
    if (!s0 && s1 || s0) F=A; else F=B;
endmodule
```



Should expand if-else into 4-bit wide multiplexor and optimize the control logic and ultimately to a single LUT on an FPGA:

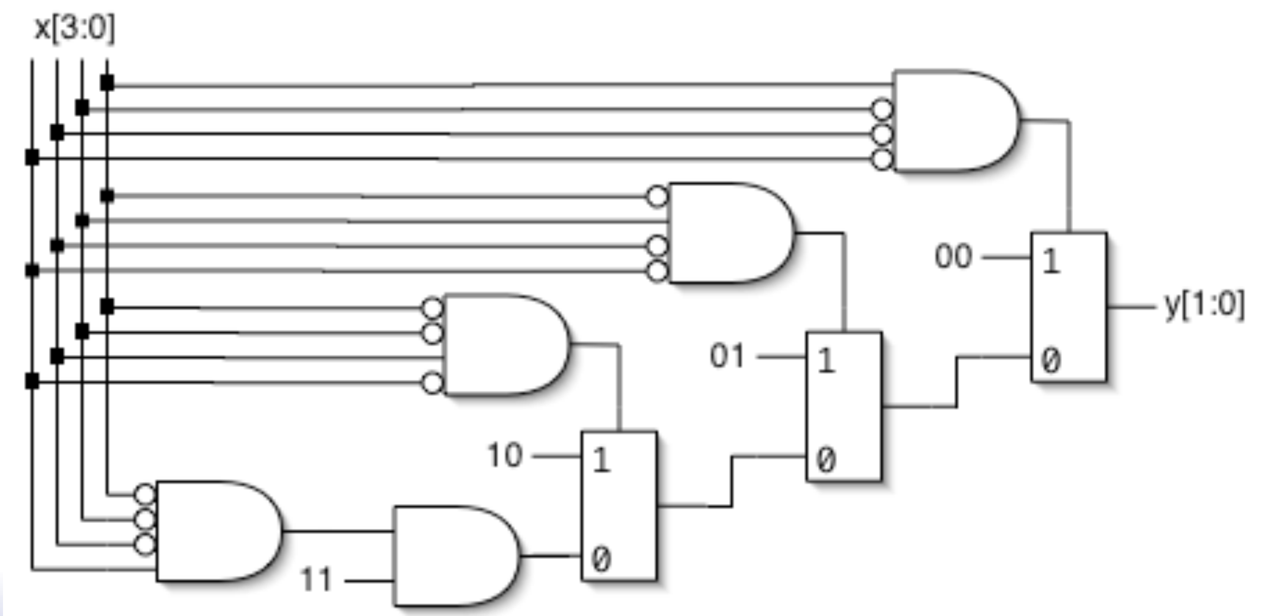
Encoder Example

Nested IF-ELSE might lead to "priority logic"

Example: 4-to-2 encoder

```
always @(x)
begin : encode
if (x == 4'b0001) y = 2'b00;
else if (x == 4'b0010) y = 2'b01;
else if (x == 4'b0100) y = 2'b10;
else if (x == 4'b1000) y = 2'b11;
else y = 2'bxx;
end
```

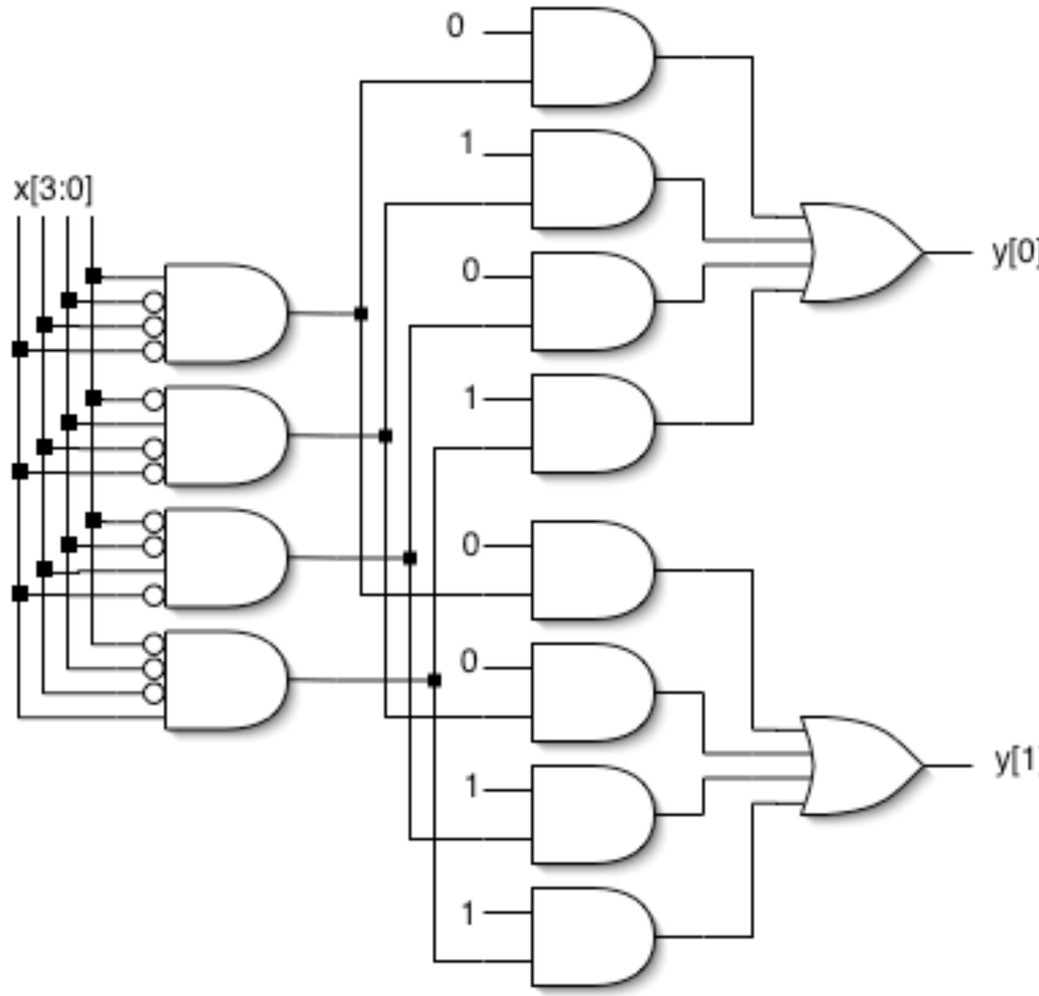
This style of cascaded logic may adversely affect the performance of the circuit.



Encoder Example (cont.)

To avoid "priority logic" use the case construct:

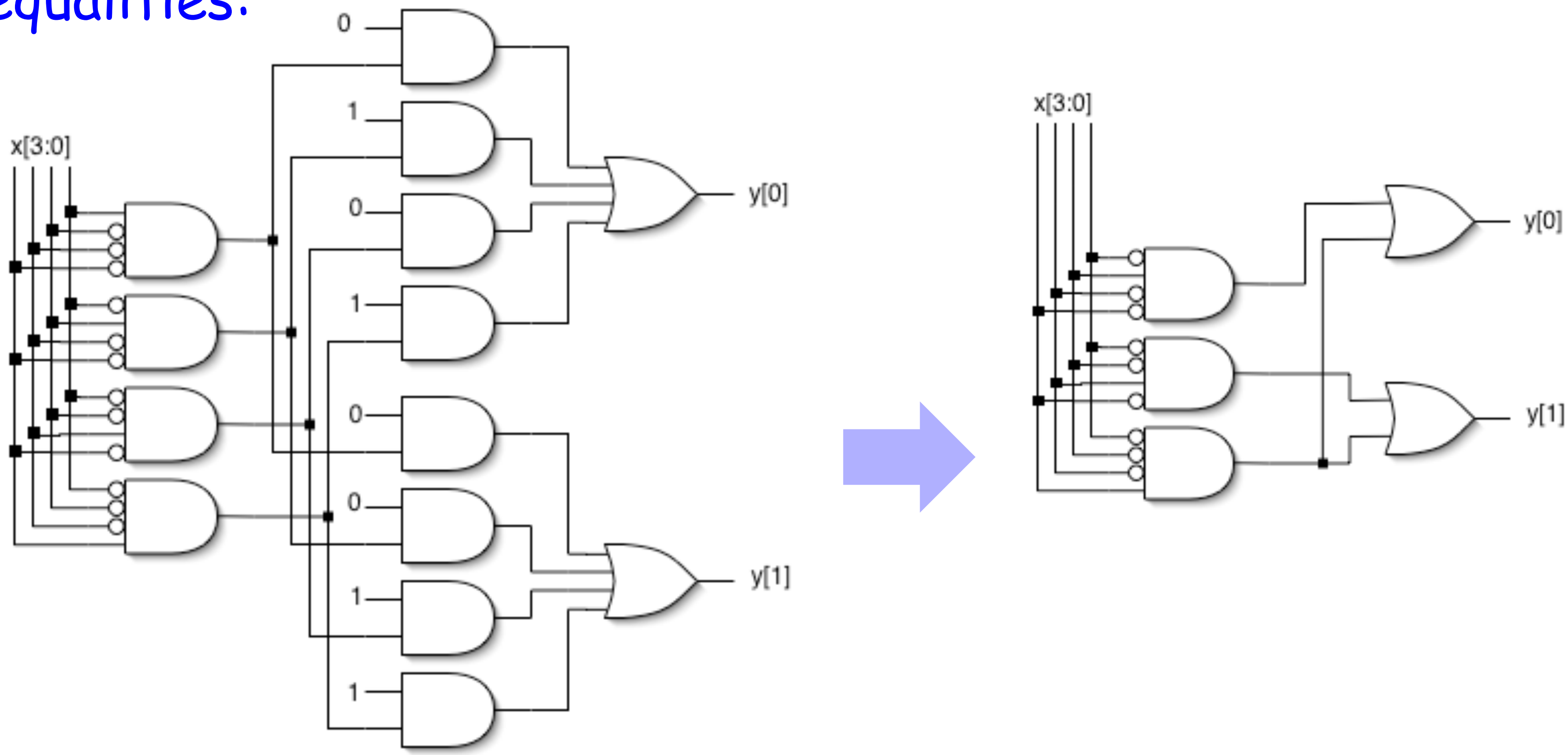
```
always @(x)
begin : encode
case (x)
4'b0001: y = 2'b00;
4'b0010: y = 2'b01;
4'b0100: y = 2'b10;
4'b1000: y = 2'b11;
default: y = 2'bxx;
endcase
end
```



All cases are matched in parallel.

Encoder Example (cont.)

This circuit would be simplified during synthesis to take advantage of constant values as follows and other Boolean equalities:



A similar simplification would be applied to the if-else version also.