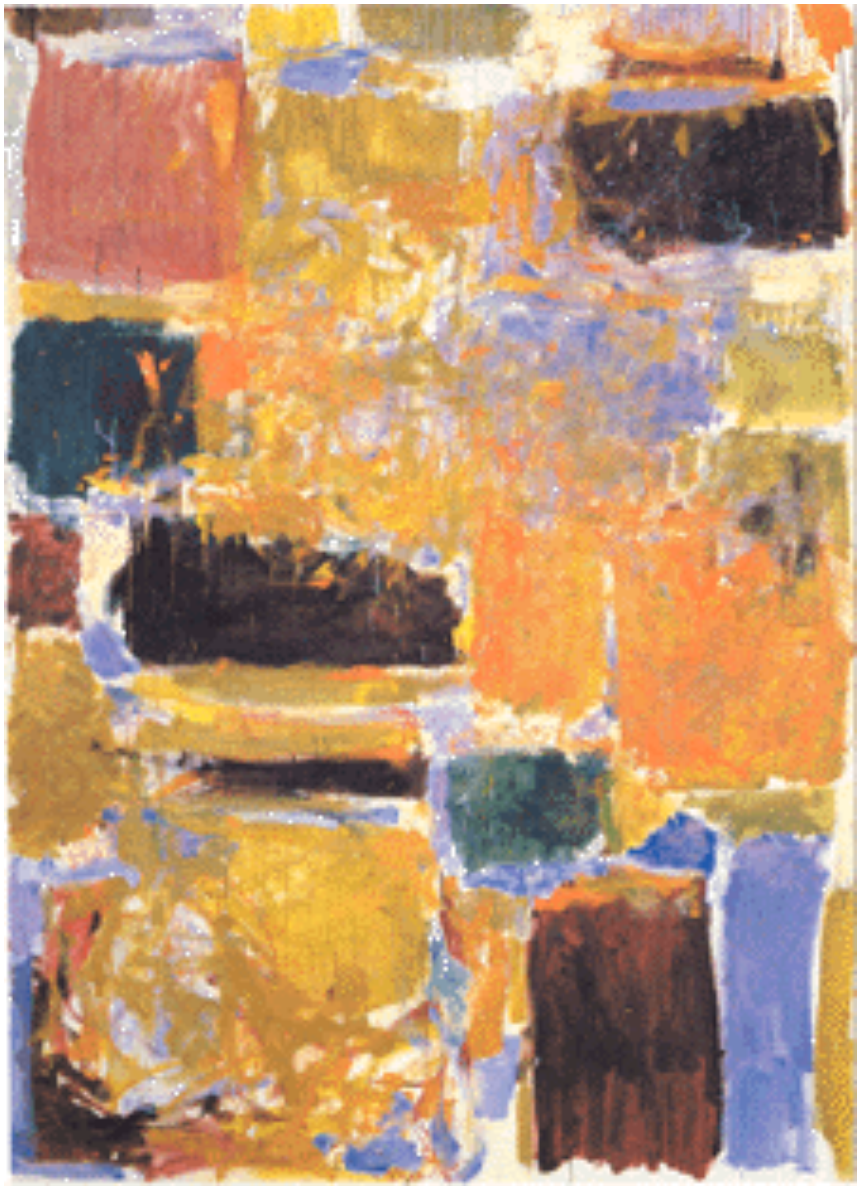
EECS 151/251A
Fall 2018
Digital Design and Integrated Circuits

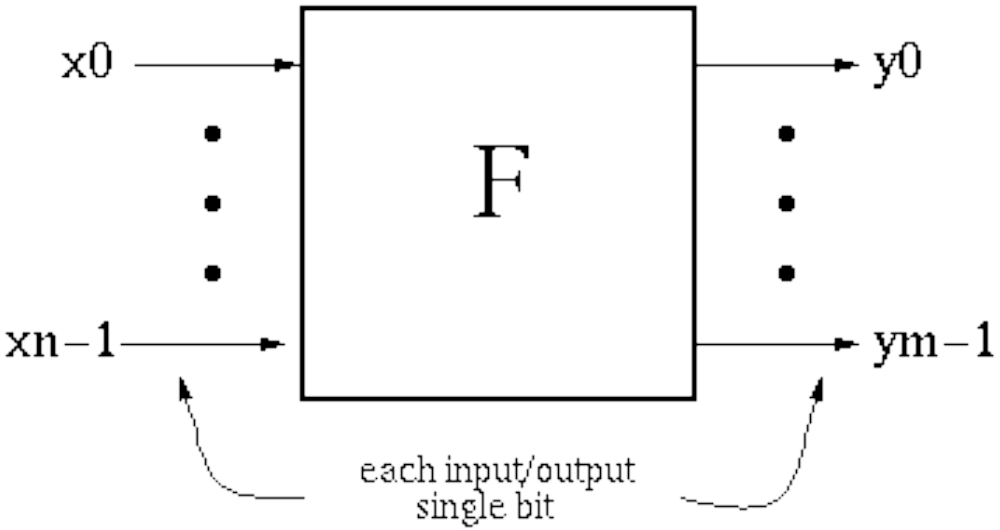
Instructor:
John Wawrzynek & Nicholas Weaver

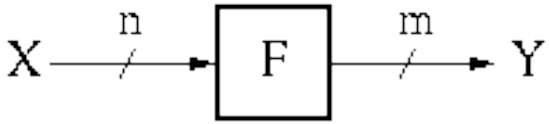
Lecture 5



Representations of Combinational Logic

Combinational Logic (CL) Defined



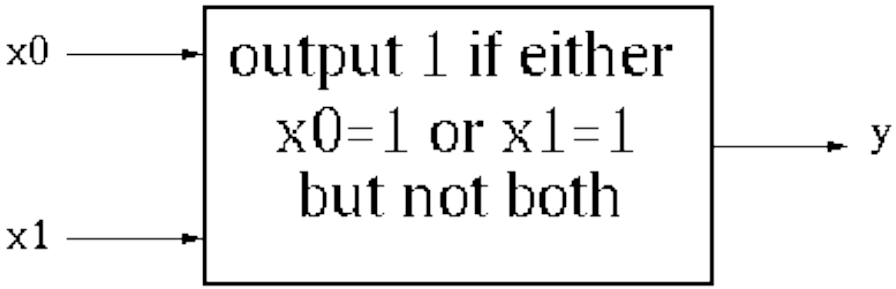


$y_i = f_i(x0 , ... , xn\text{-}1)$, where x, y are {0,1}.

Y is a function of only X.

- If we change X, Y will change immediately (well almost!).
  - There is an ***implementation dependent*** delay from X to Y.

CL Block Example #1



Boolean Equation:

$y_0 = (x_0 \text{ AND } \text{not}(x_1))$

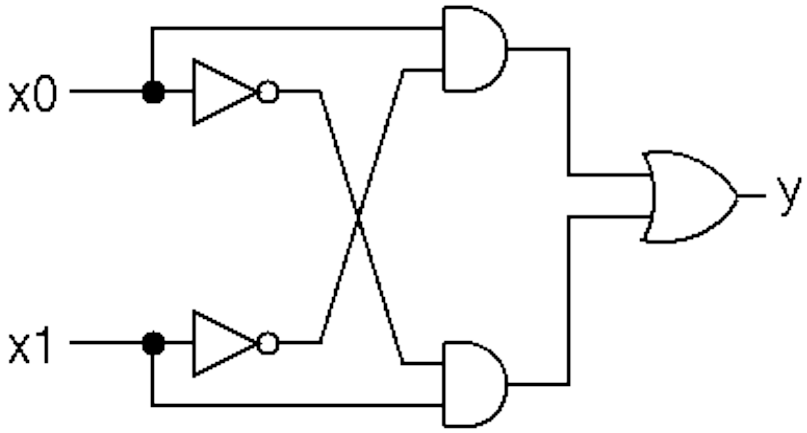
$\text{OR } (\text{not}(x_0) \text{ AND } x_1)$

$y_0 = x_0 x_1' + x_0' x_1$

Truth Table Description:

x0	x1	y
0	0	0
0	1	1
1	0	1
1	1	0

Gate Representation:

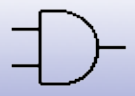


How would we prove that all three representations are equivalent?

Boolean Algebra/Logic Circuits

- Why are they called “logic circuits”?
  - Logic: The study of the principles of reasoning.
  - The 19th Century Mathematician, George Boole, developed a math. system (algebra) involving logic, Boolean Algebra.
  - His variables took on TRUE, FALSE
  - Later Claude Shannon (father of information theory) showed (in his Master's thesis!) how to map Boolean Algebra to digital circuits:
  - Primitive functions of Boolean Algebra:

a	b	AND
0	0	0
0	1	0
1	0	0
1	1	1

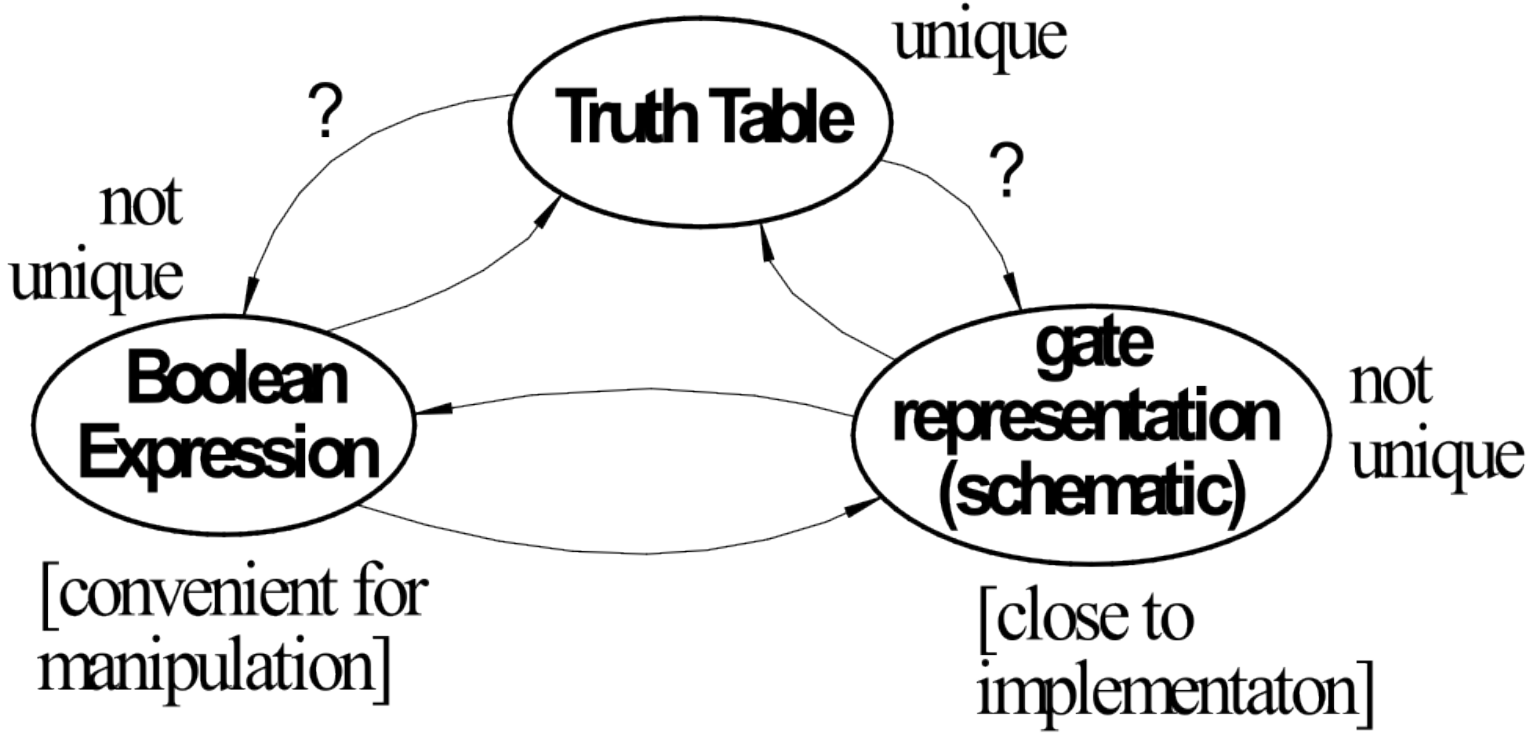


a	b	OR
0	0	0
0	1	1
1	0	1
1	1	1

a	NOT
0	0
1	0

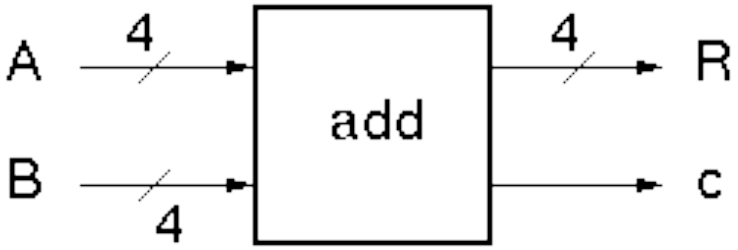
Relationship Among Representations

* 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?

CL Block Example – 4 Bit Adder



R = A + B,

c is carry out

- Truth Table Representation:

a3	a2	a1	a0	b3	b2	b1	b0	r3	r2	r1	r0	c
0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	1	0	0	0	1	0
0	0	0	0	0	0	1	0	0	0	1	0	0
0	0	0	0	0	0	1	1	0	0	1	1	0
0	0	0	0	0	1	0	0	0	1	0	0	0
•												
•												
•												
0	0	1	0	0	0	1	0	0	1	0	0	0
0	0	1	0	0	0	1	1	0	1	0	1	0
•												
•												
•												
0	0	0	1	1	1	1	1	0	0	0	0	1

In general: 2^n rows for n inputs. *256 rows!*

Is there a more efficient (compact) way to specify this function?

4-bit Adder Example

- Motivate the adder circuit design by hand addition:

```
    a3 a2 a1 a0
+   b3 b2 b1 b0
-------------
c   r3 r2 r1 r0
```

- Add a0 and b0 as follows:

a	b	r	c
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

carry to next stage

$r = a \text{ XOR } b = a \oplus b$

$c = a \text{ AND } b = ab$

```
    a3 a2 a1 a0
+   b3 b2 b1 b0
-------------
c   r3 r2 r1 r0
```

- Add a1 and b1 as follows:

ci	a	b	r	co
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

$r = a \oplus b \oplus c_i$

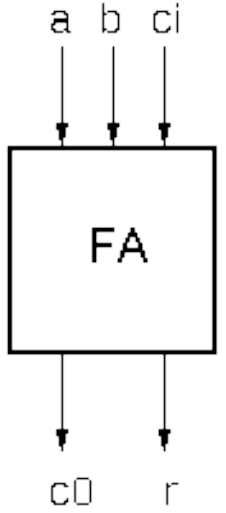
$co = ab + ac_i + bc_i$

4-bit Adder Example

- In general:

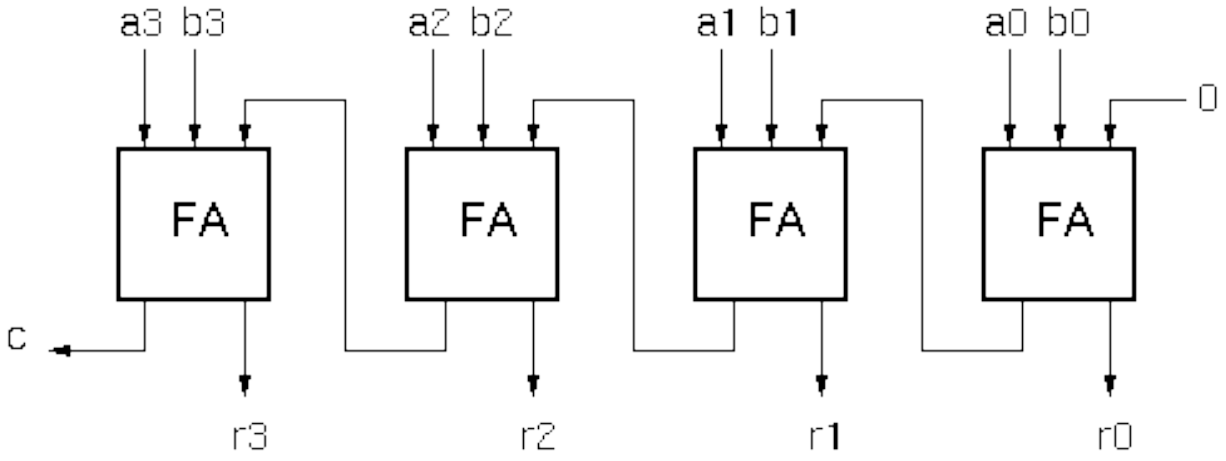
$$r_i = a_i \oplus b_i \oplus c_{in}$$

$$c_{out} = a_i c_{in} + a_i b_i + b_i c_{in} = c_{in}(a_i + b_i) + a_i b_i$$



"Full adder cell"

- Now, the 4-bit adder:



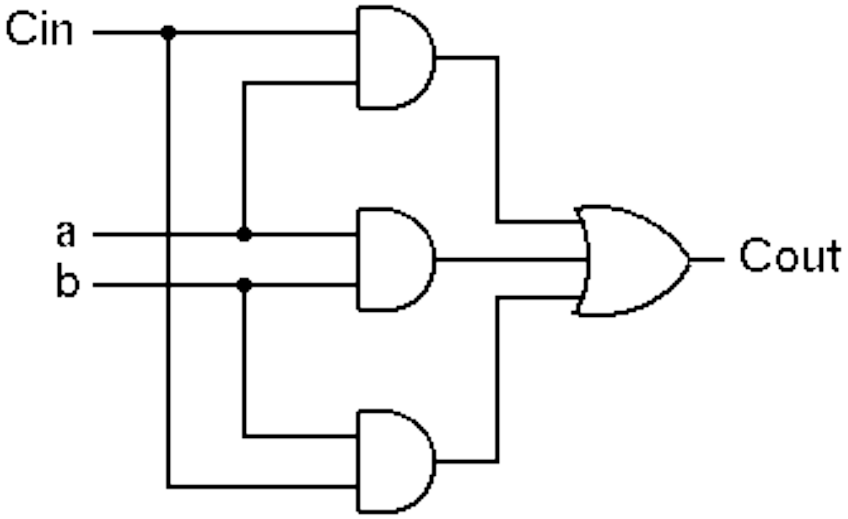
"ripple" adder

4-bit Adder Example

- Graphical Representation of FA-cell

$r_i = a_i \oplus b_i \oplus c_{in}$

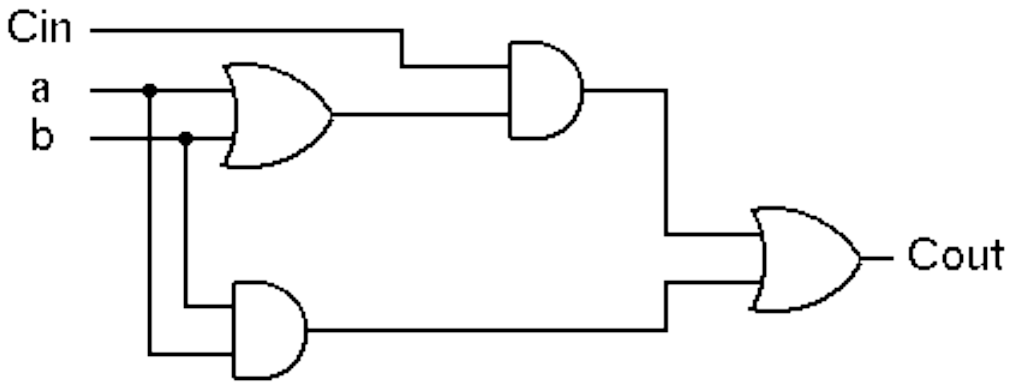
$c_{out} = a_i c_{in} + a_i b_i + b_i c_{in}$

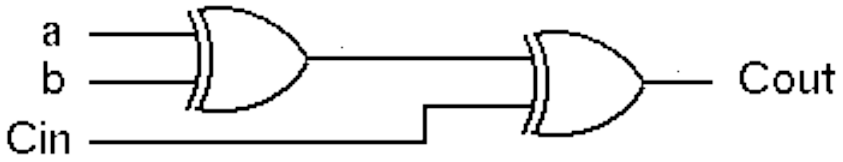


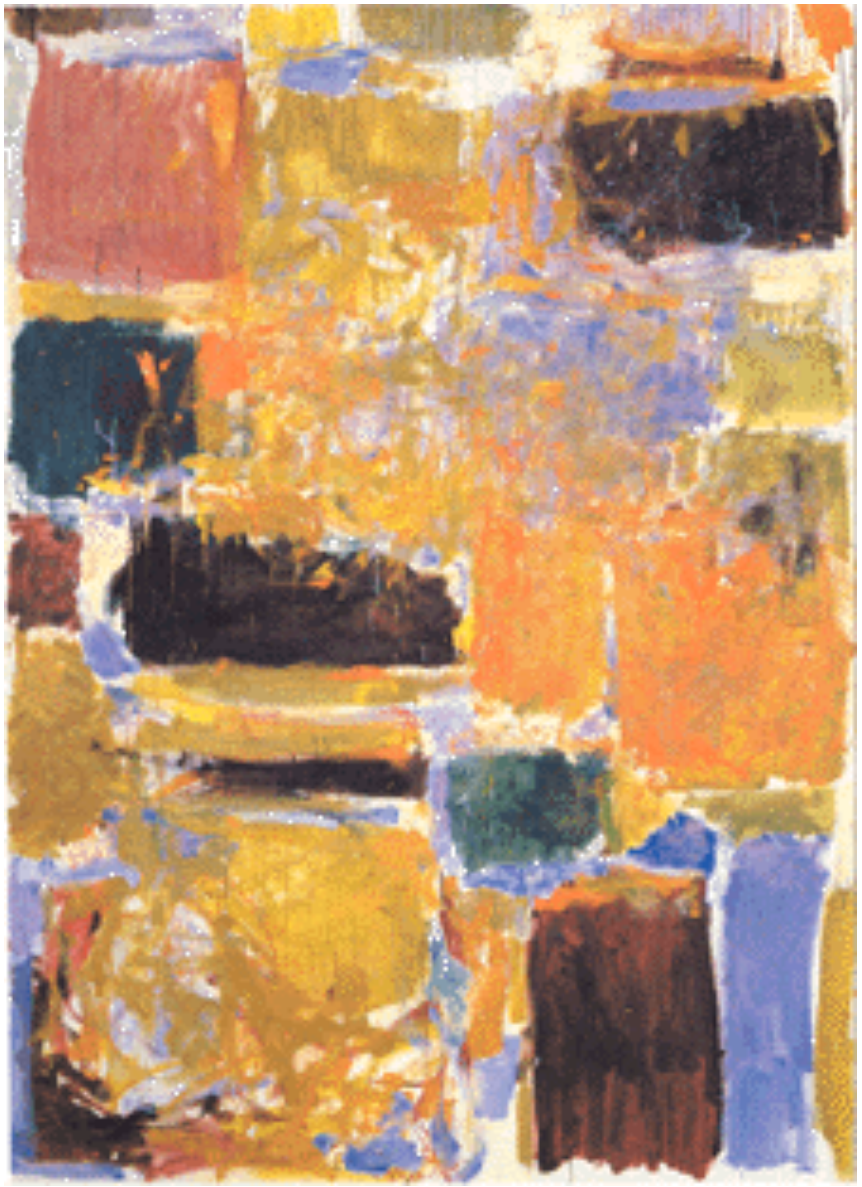
- Alternative Implementation (with only 2-input gates):

$r_i = (a_i \oplus b_i) \oplus c_{in}$

$c_{out} = c_{in}(a_i + b_i) + a_i b_i$



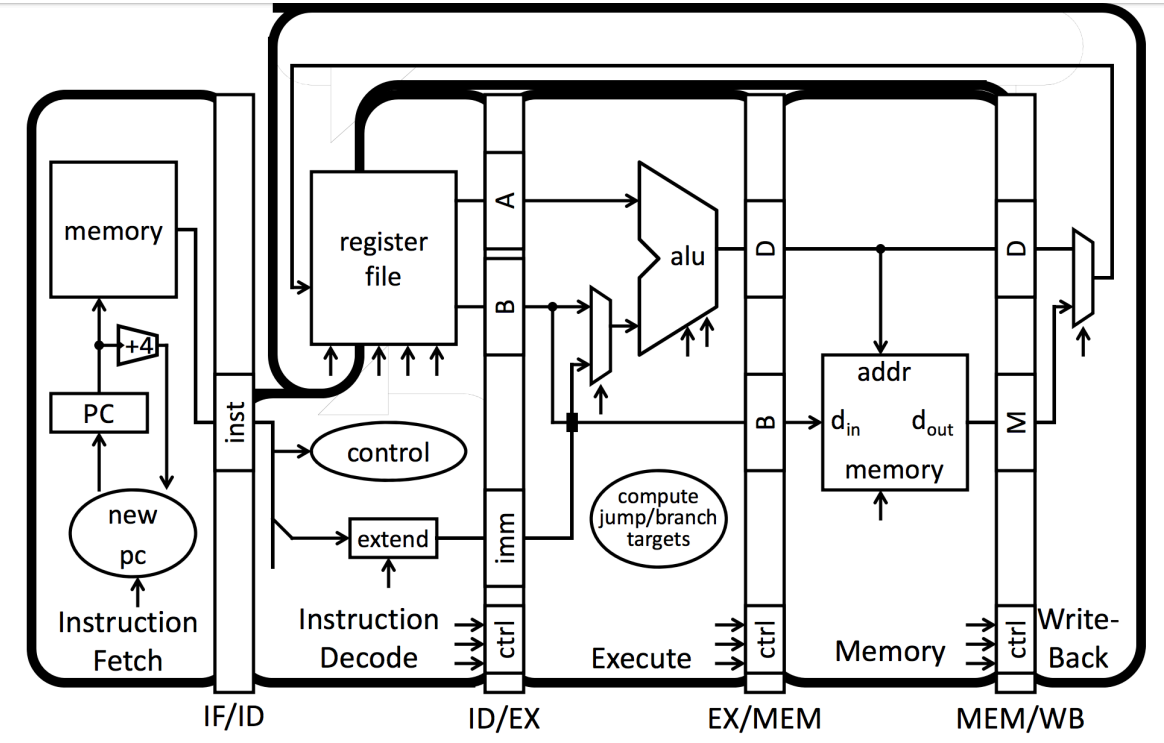




FPGAs: Directly Mapping Truth Tables

So We Want To Map Designs...

- FPGAs: Programmable fabric design to map arbitrary digital designs
- The prototypical design: A Microprocessor...



So What Do We Need?

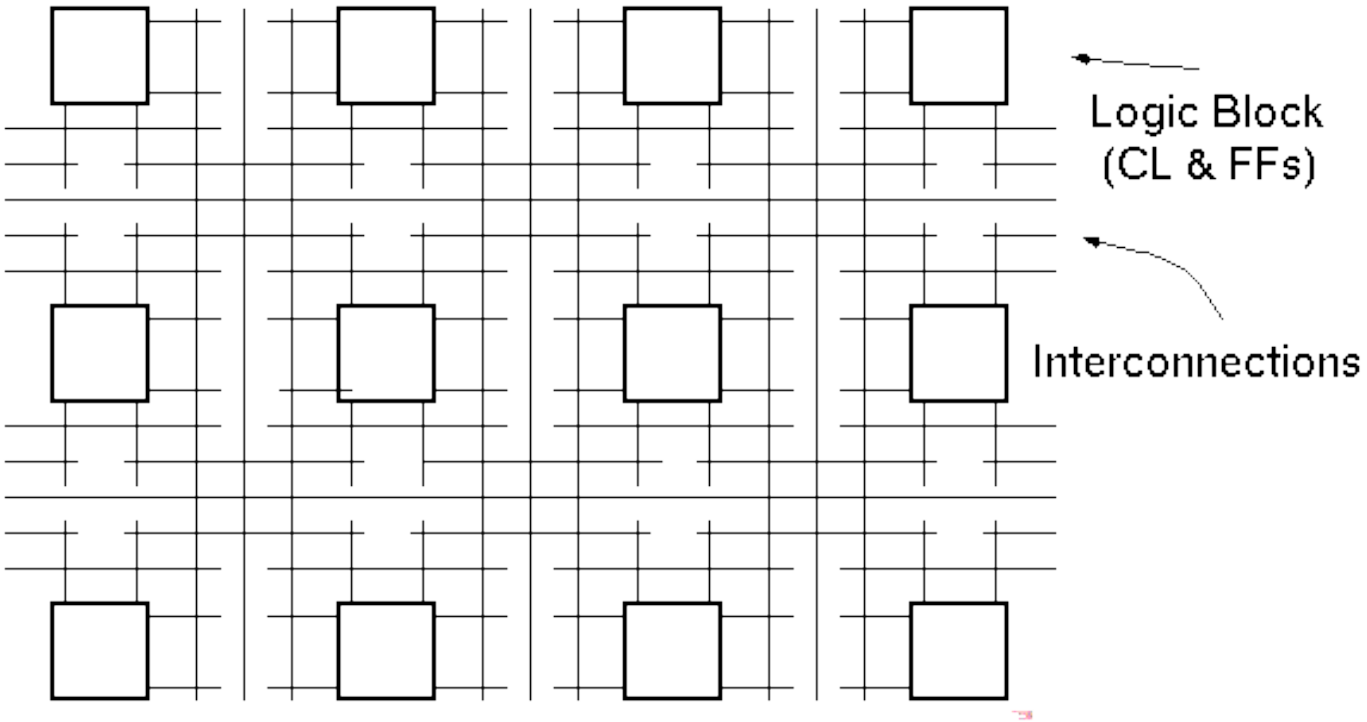
- Arbitrary boolean functions
  - For control logic
- State registers
- Multiplexors
  - The ever-present need to switch from several possible inputs
    - EG, select an add or a shift as the output of the ALU
    - Select the proper value for the forwarding
- Small memories for register files
- Larger memories for caches
- Stuff to speak to the outside world

Our FPGA Fabric: The Xilinx 7 Series

- A family of FPGAs
  - Ranging from <$20 and 900 "slices" to >$4000 and 13,600 slices (actually multiple chips)
- A common set of logic cells
  - Combinational Logic Blocks
  - Larger distributed memory blocks
  - Distributed DSP-blocks with multipliers
- A common set of tools
  - Verilog & VHDL
- Differing levels of interconnect
  - Larger FPGAs need more wires per unit of logic

FPGA Overview

- Basic idea: two-dimensional array of logic blocks and flip-flops with a means for the user to configure (program):
    1. the interconnection between the logic blocks,
    2. the function of each block.



Simplified version of FPGA internal architecture

Why are FPGAs Interesting?

- Technical viewpoint:
  - For hardware/system-designers, like ASICs - only better: "Tape-out" new design every few minutes/ hours.
  - "reconfigurability" or "reprogrammability" may offer other advantages over fixed logic?
    - In-field reprogramming? Dynamic reconfiguration? Self-modifying hardware, evolvable hardware?
  - Coupled to large & useful fixed logic
    - Processor/FPGA hybrids
- Cost viewpoint:
  - Not only is the design cost vastly less
  - The manufacturers produce a ***lot*** of these things

Why are FPGAs Interesting?

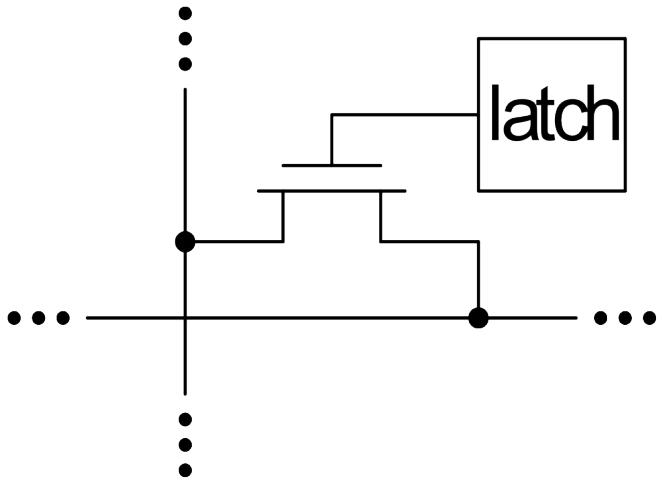
- Staggering logic capacity growth (10,000x):

Year Introduced	Device	Logic Cells	"logic gate equivalents"
1985	XC2064	128	1024
2015	XCVU37P	2.8M	10M+
1995 EECS150	XC4005	196	5000
2015 EECS151	XCZ7020	85K	....

  - FPGAs have tracked Moore's Law better than any other programmable device.
    - Effectively almost perfectly: Just keep adding more transistors
  - Plus they are also a lot faster...

User Programmability

- Latch-based (Xilinx, Altera, ...)



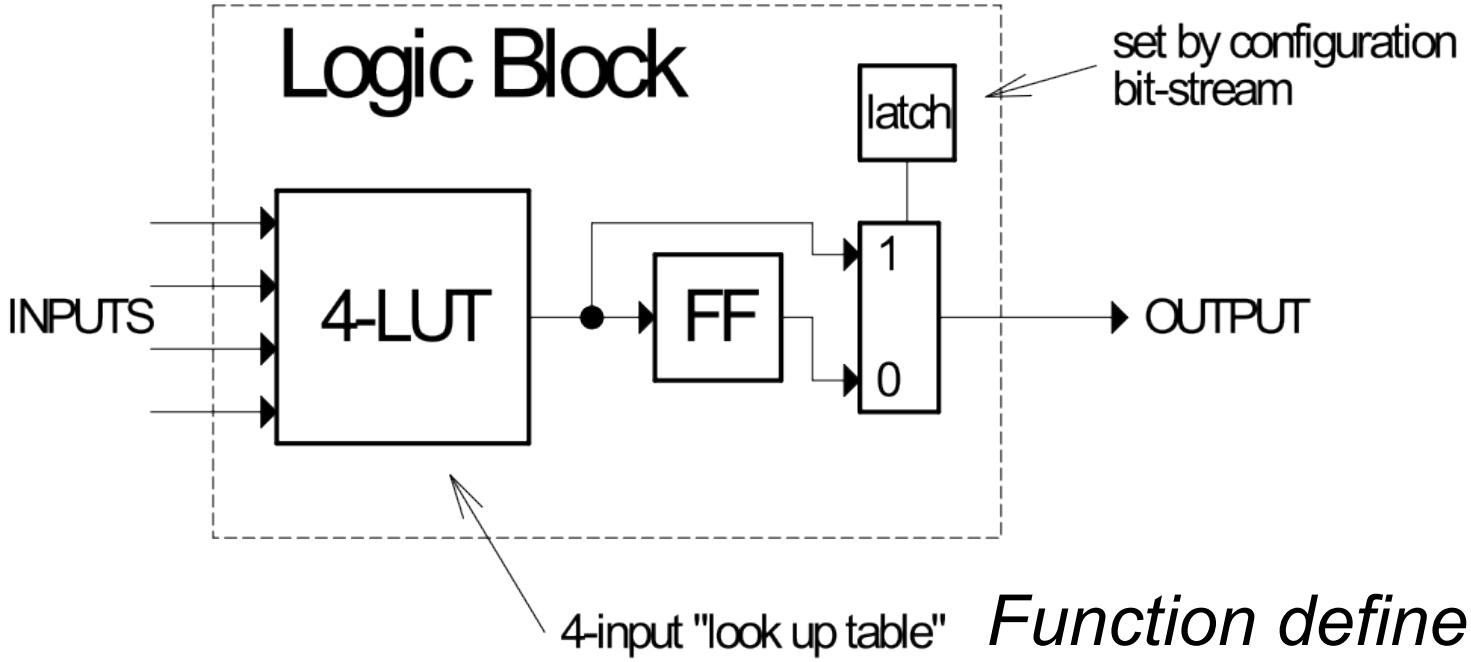
\+ reconfigurable

– volatile

– relatively large.

- Latches are used to:
  1. control a switch to make or break cross-point connections in the interconnect
  2. define the function of the logic blocks
  3. set user options:
     - within the logic blocks
     - in the input/output blocks
     - global reset/clock
- "Configuration bit stream" is loaded under user control: Shift in the programming data

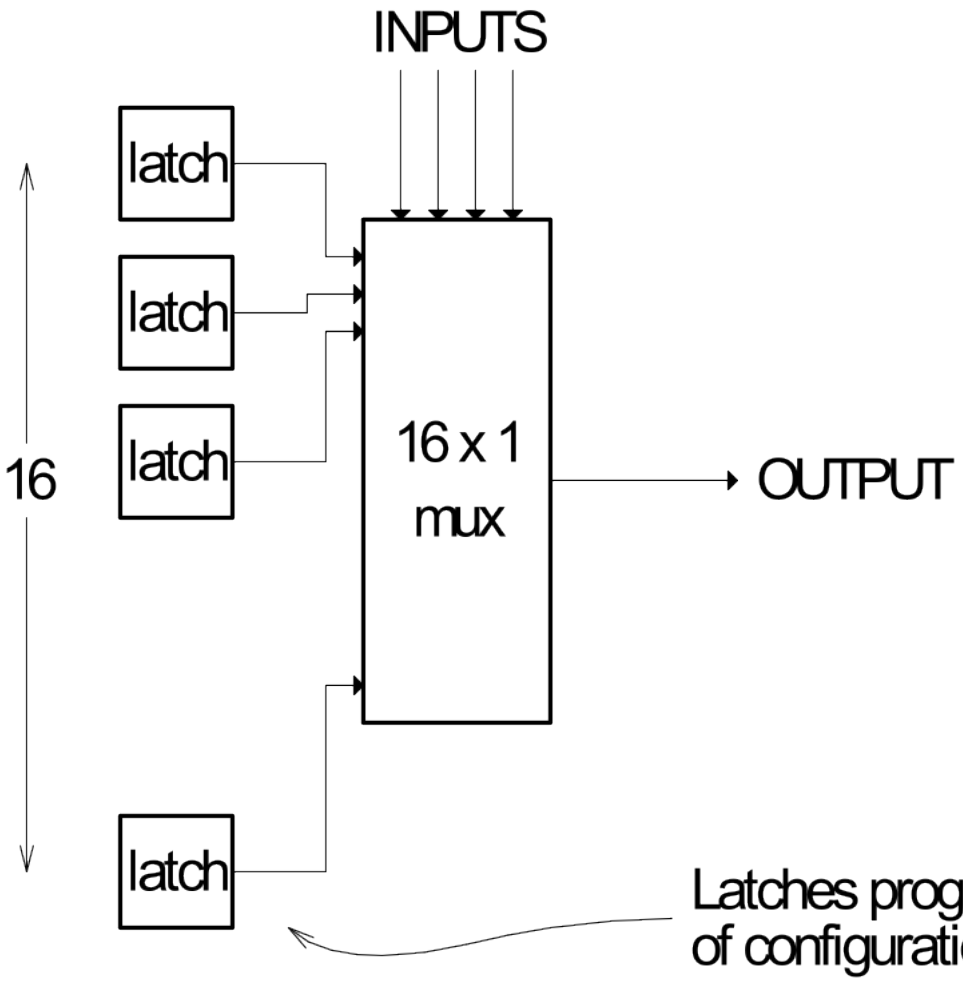
Idealized FPGA Logic Block



Function defined by configuration bit-stream

- 4-input look up table (LUT)
  - implements combinational logic functions
- Register
  - optionally stores output of LUT

4-LUT Implementation



- n-bit LUT is implemented as a 2^n x 1 memory:
  - inputs choose one of 2^n memory locations.
  - memory locations (latches) are normally loaded with values from user's configuration bit stream.
  - Inputs to mux control are the CLB inputs.
- Result is a general purpose "logic gate".
  - n-LUT can implement any function of n inputs by simply ***directly*** implementing the truth table representation

LUT as general logic gate

- An n-lut as a direct implementation of a function truth-table.
- Each latch location holds the value of the function corresponding to one input combination.

Example: 2-lut

INPUTS	AND	OR
00	0	0
01	0	1
10	0	1
11	1	1

• • •

Implements any function of 2 inputs.

How many of these are there?

How many functions of n inputs?

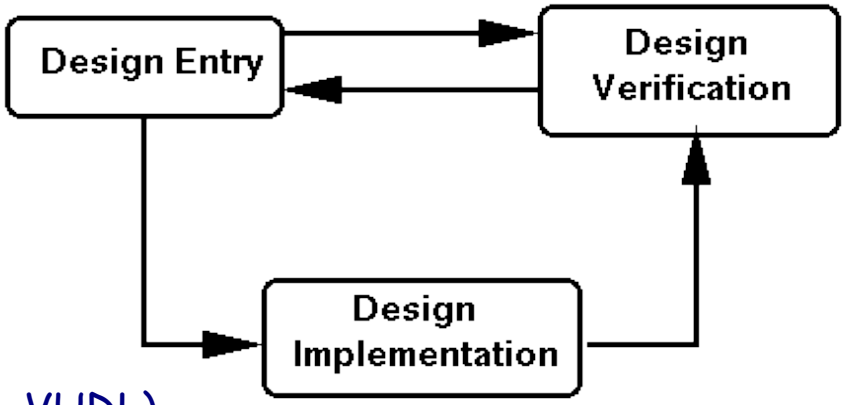
Example: 4-lut

INPUTS	
0000	F(0,0,0,0) ← store in 1st latch
0001	F(0,0,0,1) ← store in 2nd latch
0010	F(0,0,1,0) ←
0011	F(0,0,1,1) ←
0011	
0100	•
0101	•
0110	•
0111	
1000	
1001	
1010	
1011	
1100	
1101	
1110	
1111	

How Big A LUT...

- The original FPGAs used 4-input LUTs
  - Able to implement an arbitrary 4 input binary function
- But what's the most common function?
  - A mux...
- A 4-LUT can only implement a 2-1 MUX
  - Additional logic around the LUTs to build larger MUXes
- A 6-LUT can implement a 4-1 MUX...
  - So in the past few years the FPGA primary architectures have switched from 4-LUTs to 6-LUTs
  - Well, actually paired 5-LUTs:
    Two 5-LUTs with common inputs and a final multiplexor

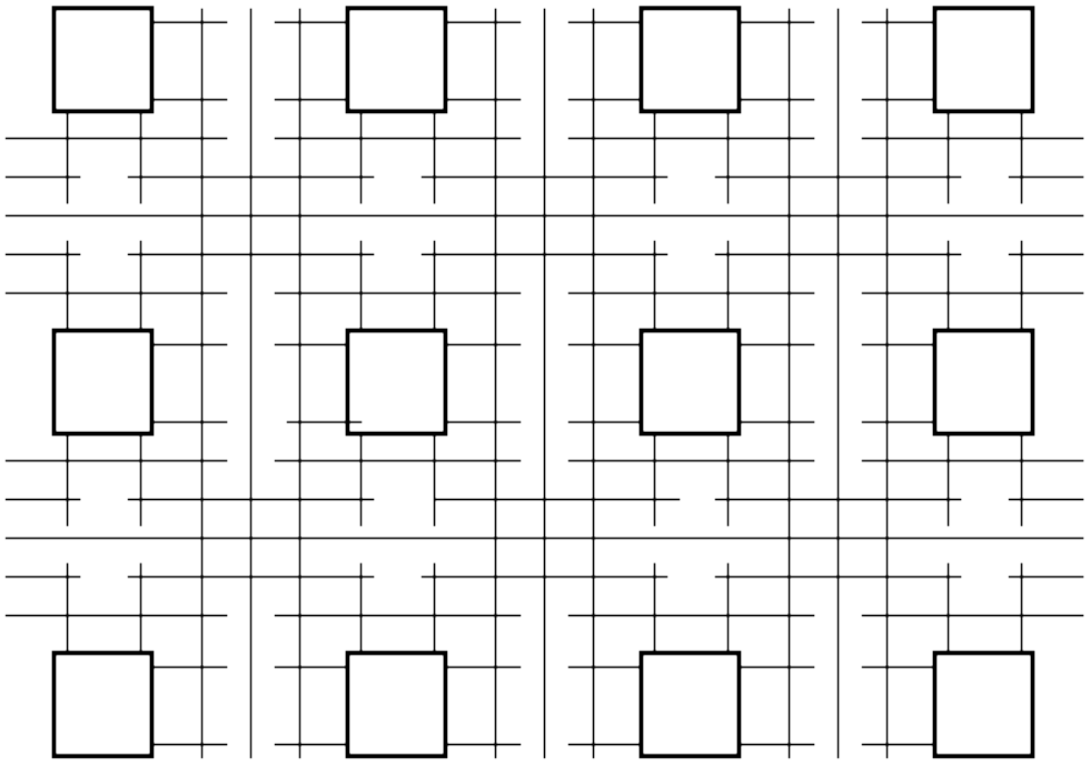
FPGA Generic Design Flow



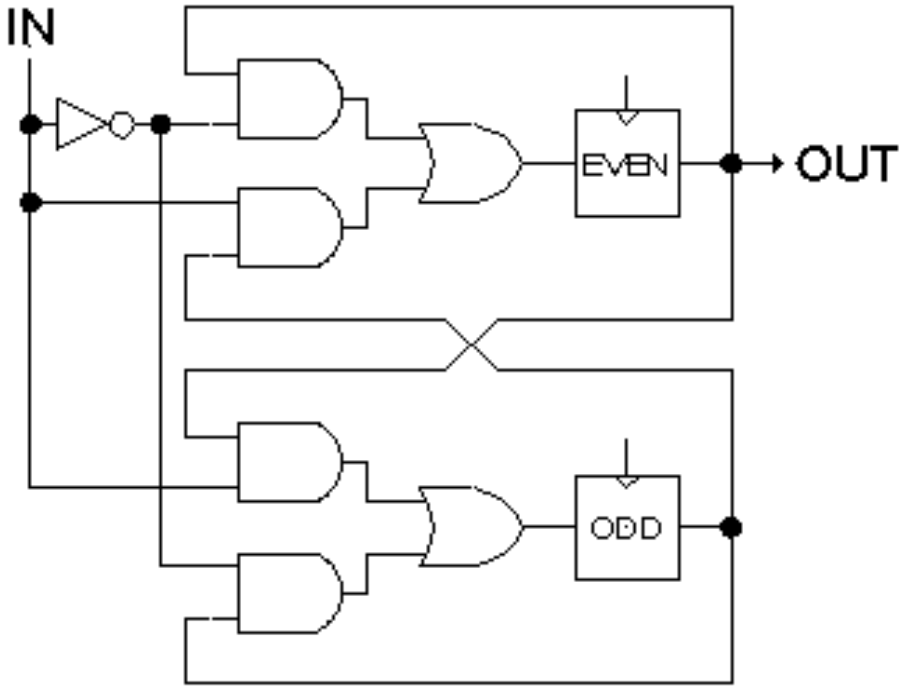
- Design Entry:
  - Create your design files using:
    - schematic editor or
    - HDL (hardware description languages: Verilog, VHDL)
- Design Implementation:
  - Logic synthesis (in case of using HDL entry) followed by,
  - Partition, place, and route to create configuration bit-stream file
- Design verification:
  - Optionally use simulator to check function,
  - Load design onto FPGA device (cable connects PC to development board), optional "logic scope" on FPGA
    - check operation at full speed in real environment.

Example Partition, Placement, and Route

- Idealized FPGA structure:



- Example Circuit:
  - collection of gates and flip-flops



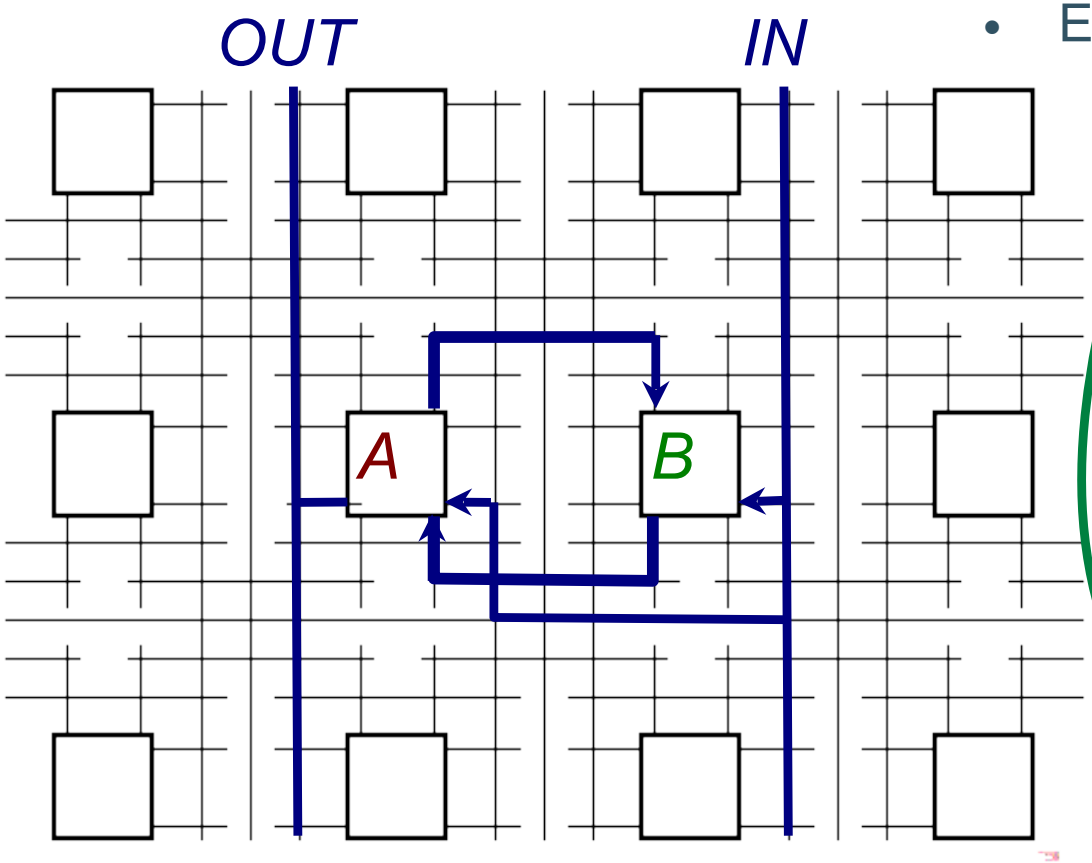
Circuit combinational logic must be "covered" by 4-input 1-output LUTs.

Flip-flops from circuit must map to FPGA flip-flops.
(Best to preserve "closeness" to CL to minimize wiring.)

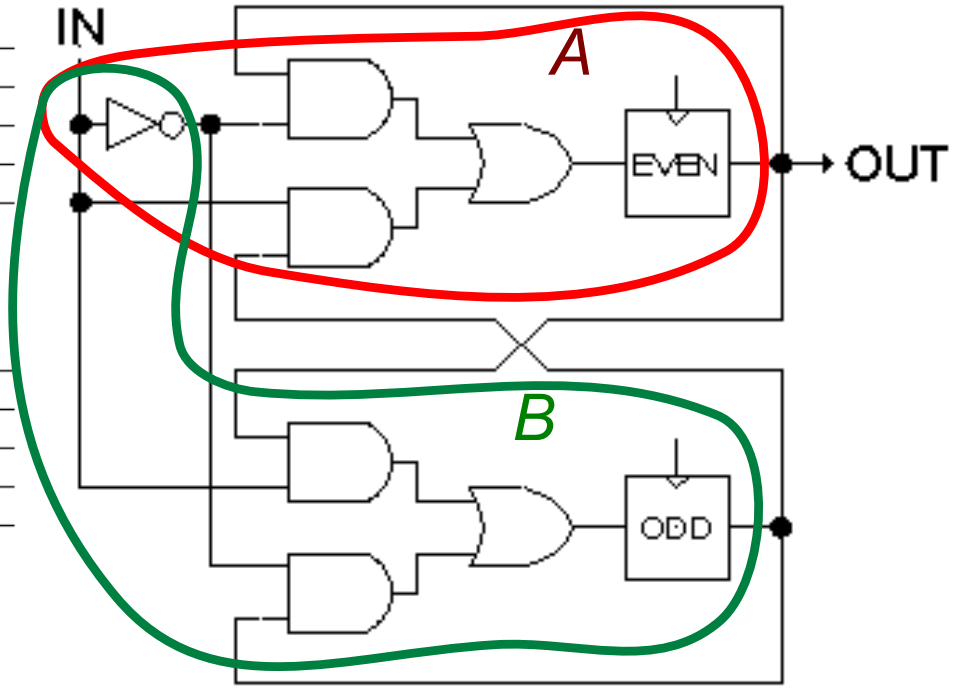
Best placement in general attempts to minimize wiring.

Vdd, GND, clock, and global resets are all "prewired".

Example Partition, Placement, and Route



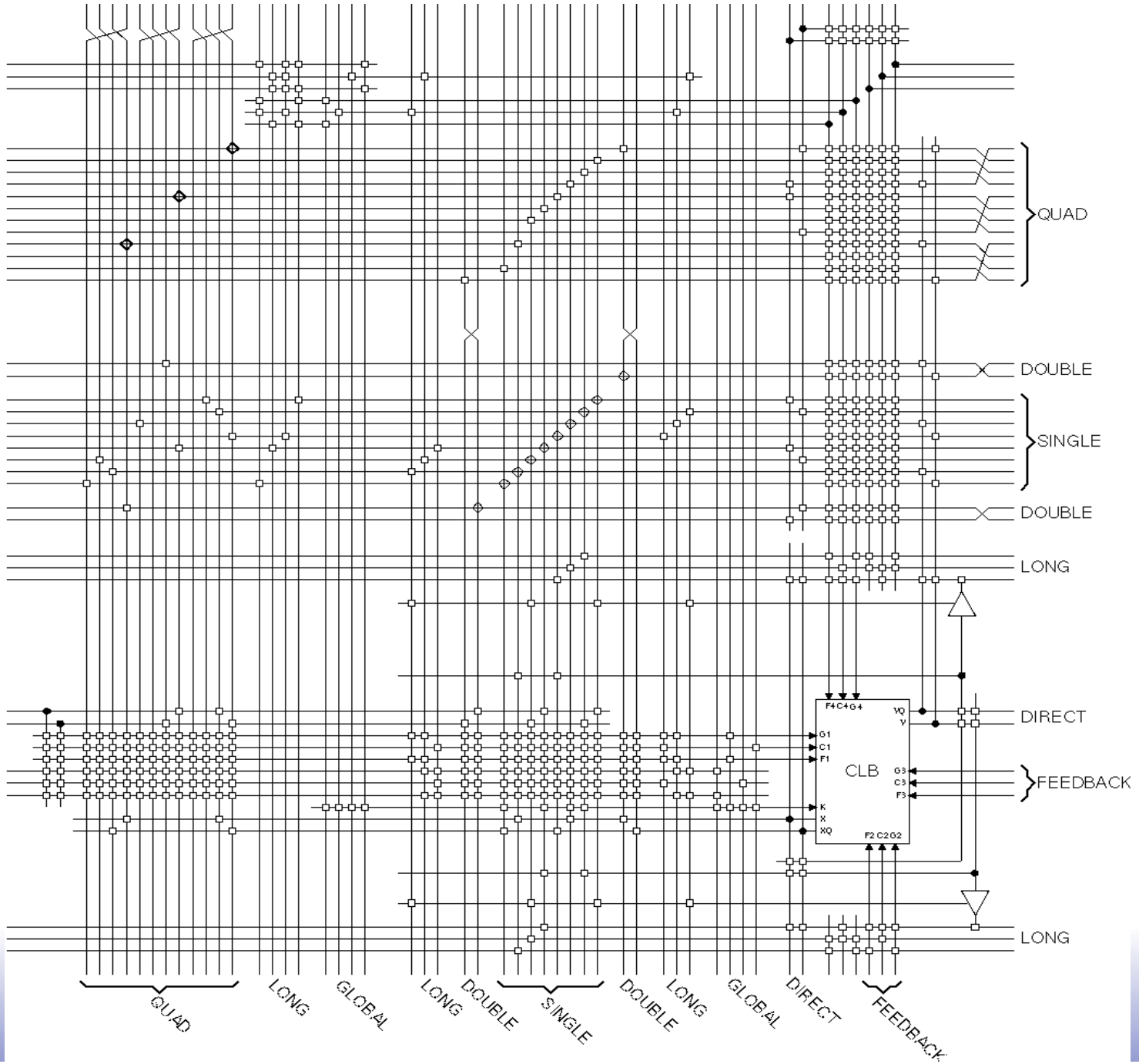
- Example Circuit:
  - collection of gates and flip-flops



Two partitions. Each has single output, no more than 4 inputs, and no more than 1 flip-flop. In this case, inverter goes in both partitions.

Note: the partition can be arbitrarily large as long as it has not more than 4 inputs and 1 output, and no more than 1 flip-flop.

Xilinx FPGAs (interconnect detail): OLD SCHOOL



Xilinx 7-Series CLB "Slice"

- Two types of slices:
  - SLICEL: Logic Functionality
  - SLICEM: + Memory Functionality
  - Two slices in a Combinational Logic Block
- Arranged in columns with other functionality
  - DSP slices: Multiply/Accumulate logic (you have 120 of them if you are in the FPGA lab)
  - Larger 36 Kb, dual-ported distributed memories
  - I/O pins
  - Hard modules for serial communication, analog input, processor, abstract DRAM interface etc

The SLICEL and SLICEM

- 8 flip/flops
  - Common clock/CE/RST line
- Each LUT
  - 6-LUT
  - 2x 5-LUT with common inputs
  - *32b memory*
  - *32b shift register*
- Dedicated carry chain
  - FPGA interconnect is slow

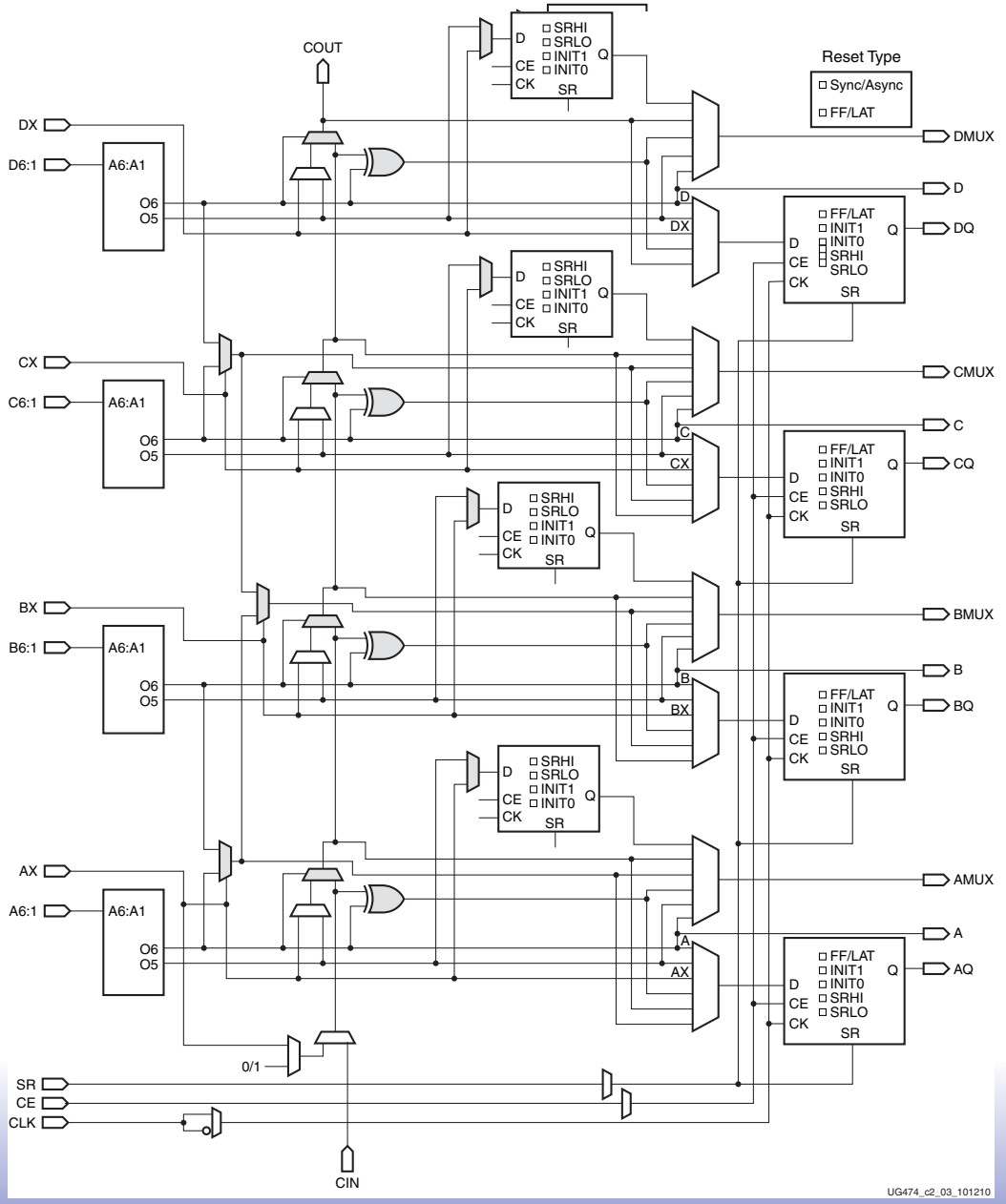
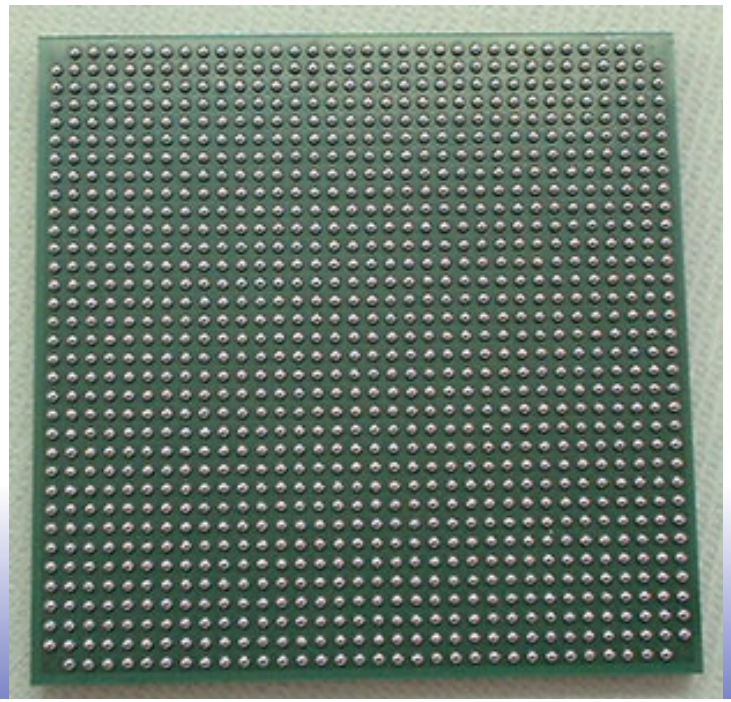
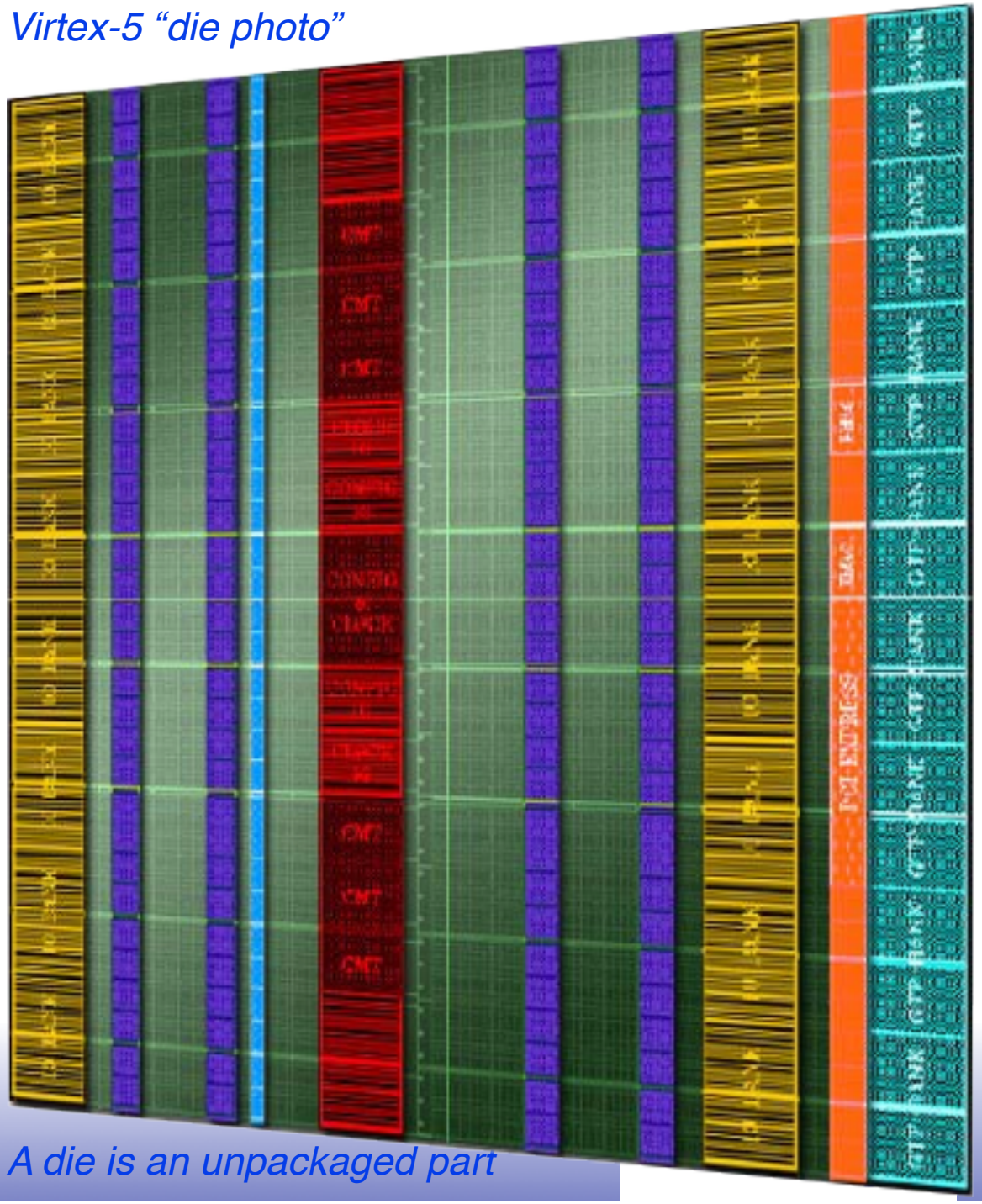


Figure 2-3: **Diagram of SLICEM**
Figure 2-4: **Diagram of SLICEL**

Xilinx Virtex-5 XC5VLX110T



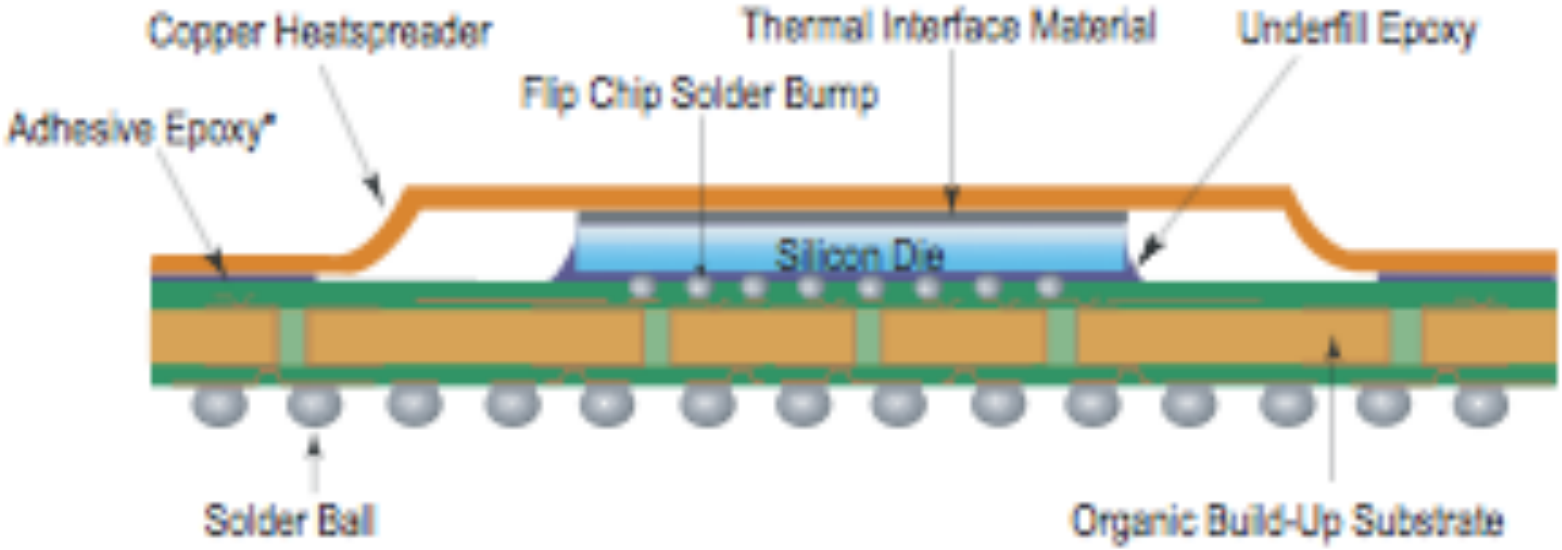
Virtex-5 “die photo”

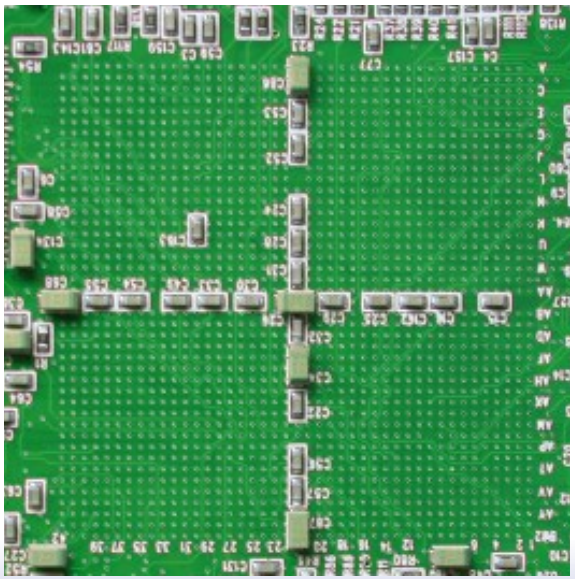


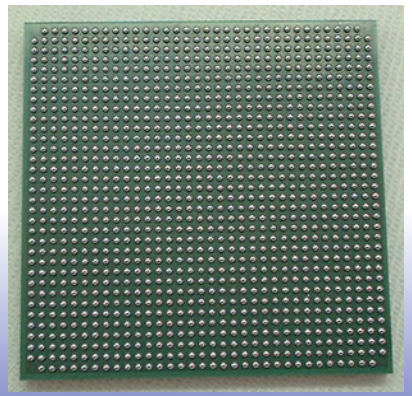
A die is an unpackaged part

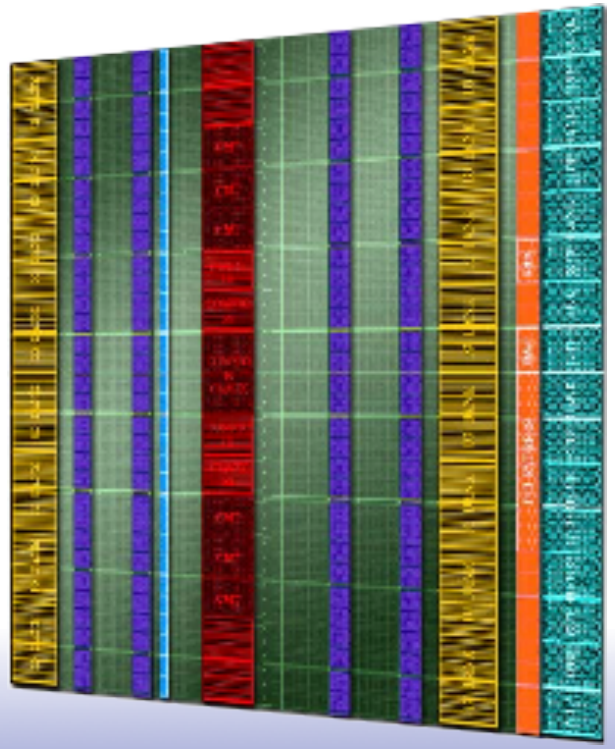
From die to PC board ...

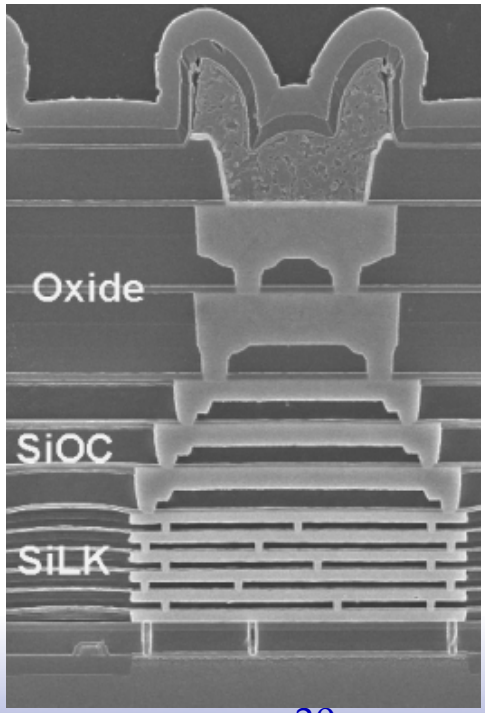
Ball Grid Array (BGA) Flip-Chip Package





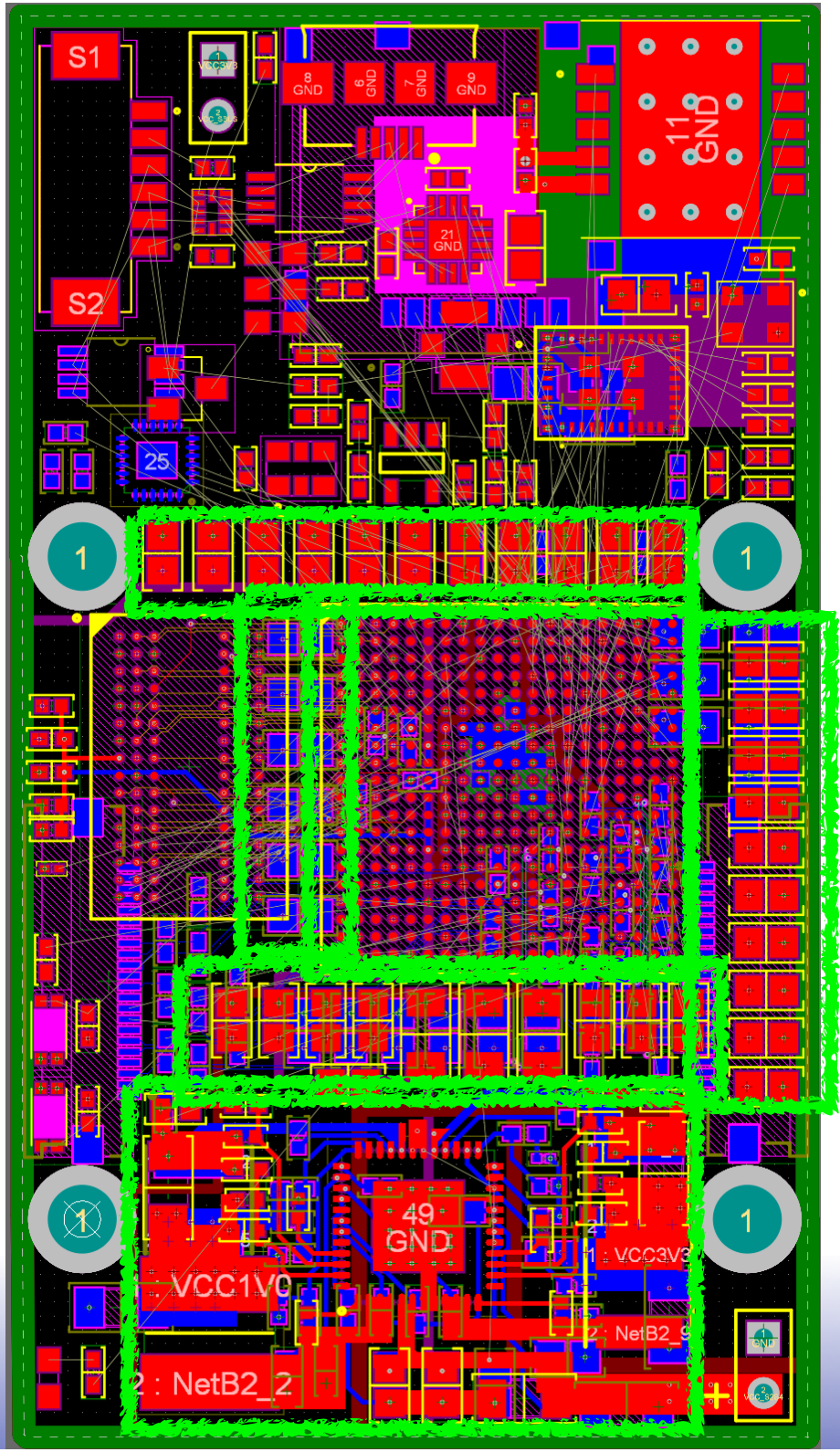


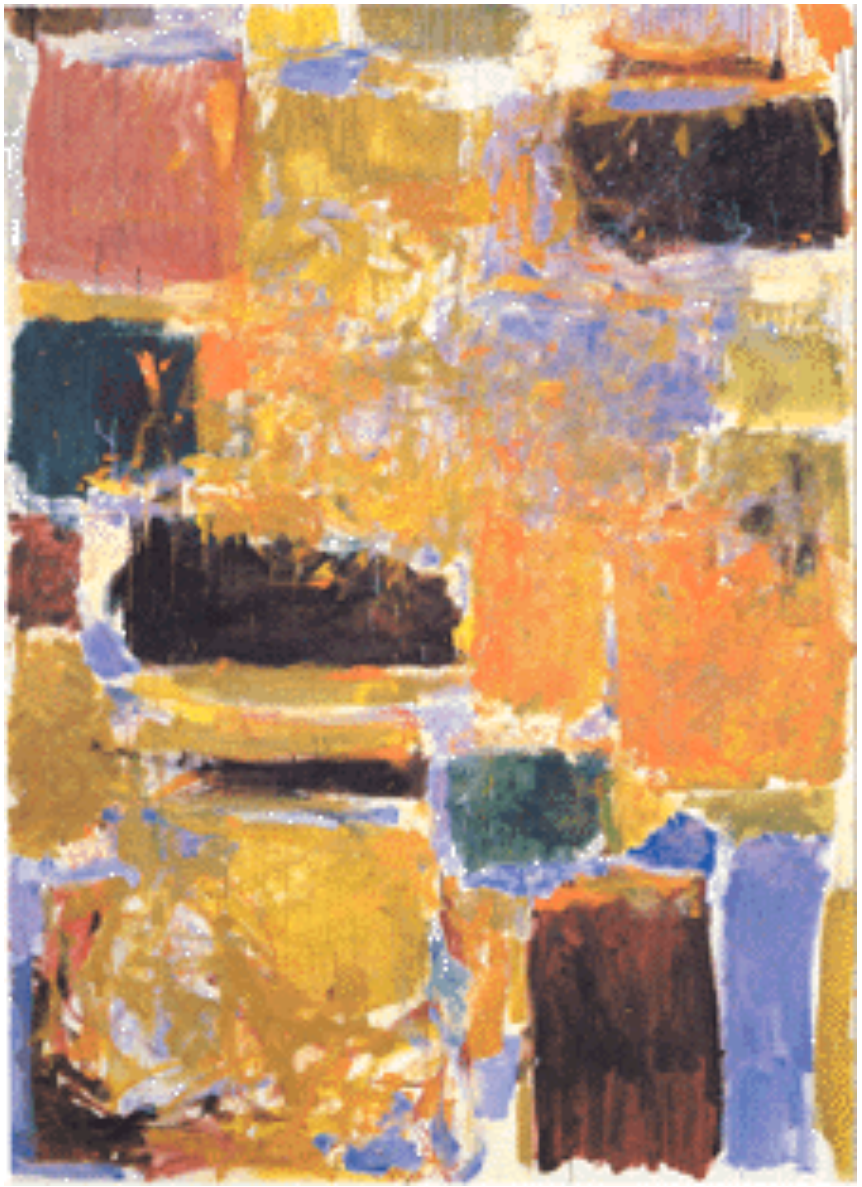




Perhaps the Biggest Problem with FPGAs: POWER

- Not only is the interconnect and gates somewhat slower than an ASIC...
  - They have a ***lot*** more capacitance and therefore burn a lot more power
- Plus multiple voltages
  - 1.0V core
  - 1.5V DDR3
  - 1.8V high speed I/O
  - 3.3V lower speed I/O
- It isn't just the power supply...
  - But also the need for ***bypass capacitors***: Temporary energy supplies to provide a stable power supply





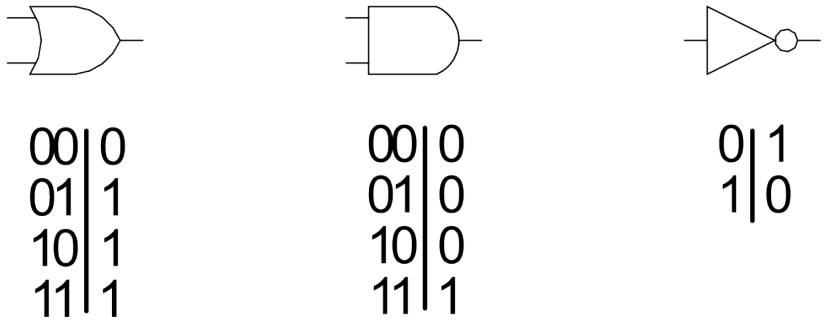
Boolean Algebra

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



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).

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

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)
```

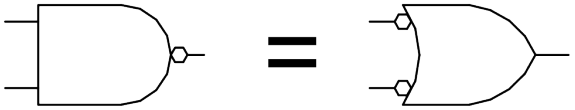
DeMorgan's Law

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

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

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

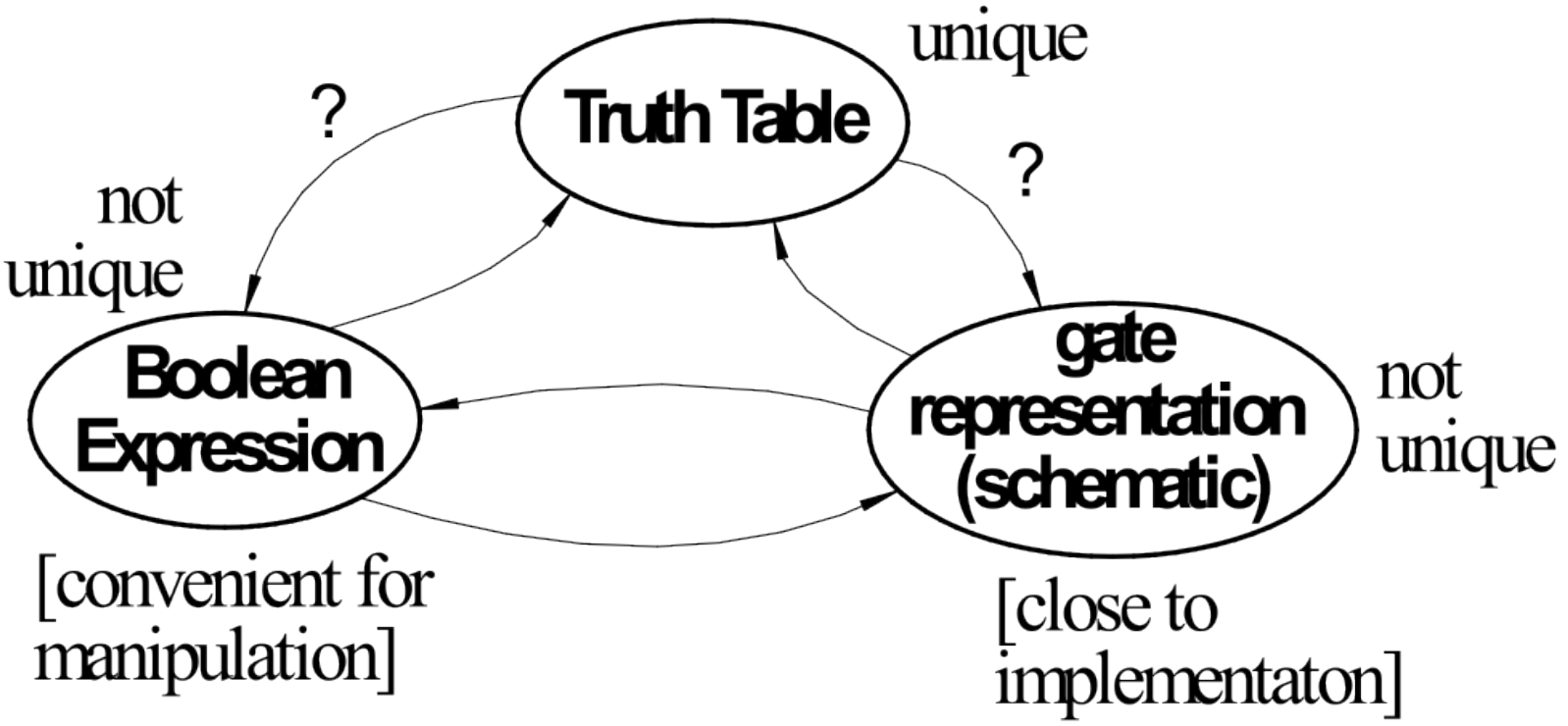


Exhaustive Proof

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:

```
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
```

```
f' = a'b'c' + a'b'c + a'bc'
   = a'b' + a'bc'
   = a' ( b' + bc' )
   = a' ( b' + c' )
   = a'b' + a'c'
```

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

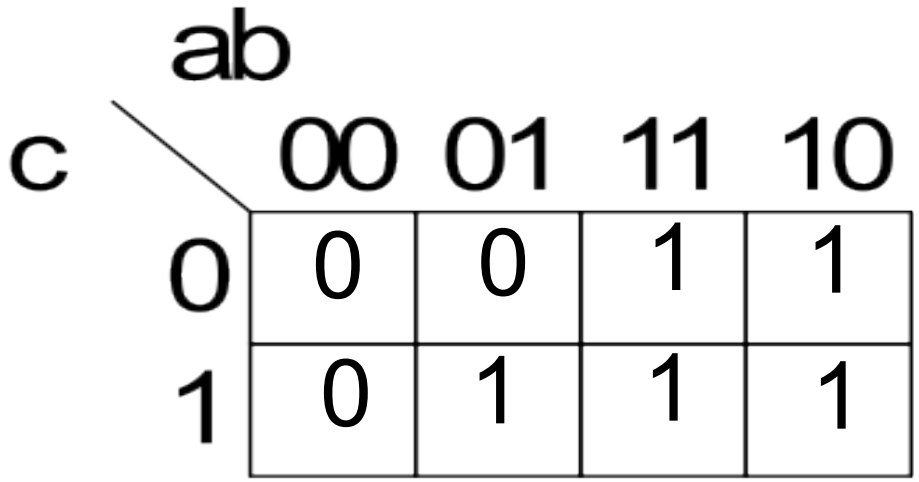
*One sum (**or**) term for each **0** in f:*

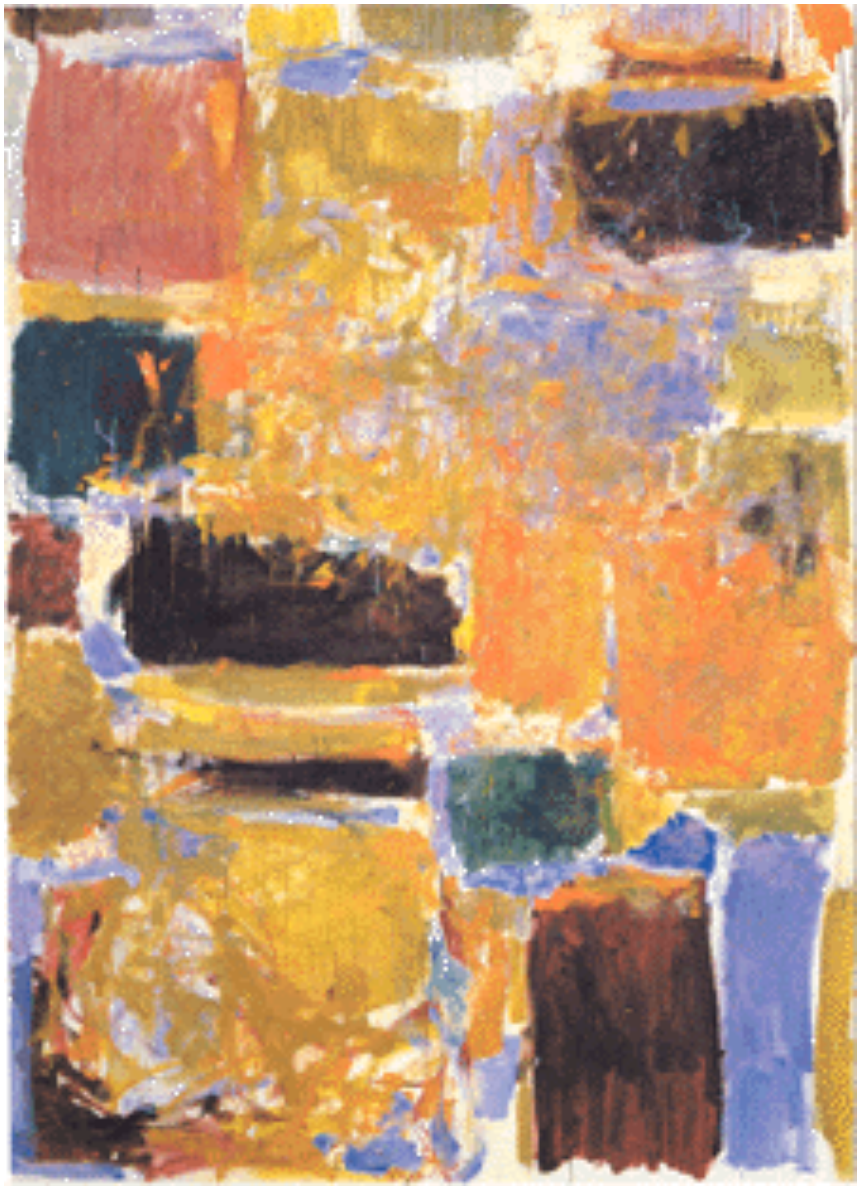
$f = (a+b+c)(a+b+c')(a+b'+c)$

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

Simplify … algebra or K-maps

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





Multi-level Logic

Multi-level Combinational Logic

- Example: reduced sum-of-products form

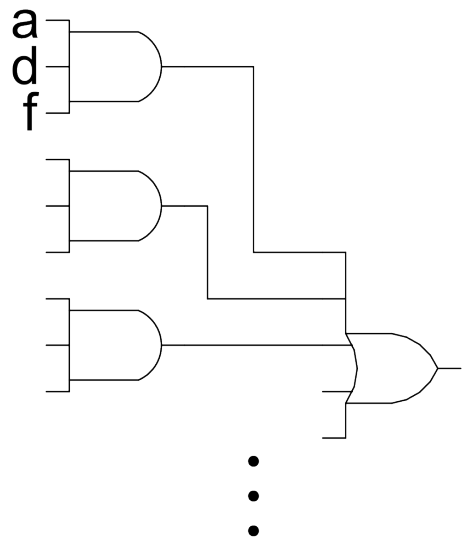
  `x = adf + aef + bdf + bef + cdf + cef + g`

- Implementation in 2-levels with gates:

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

  => 50 transistors

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



Multi-level Combinational Logic

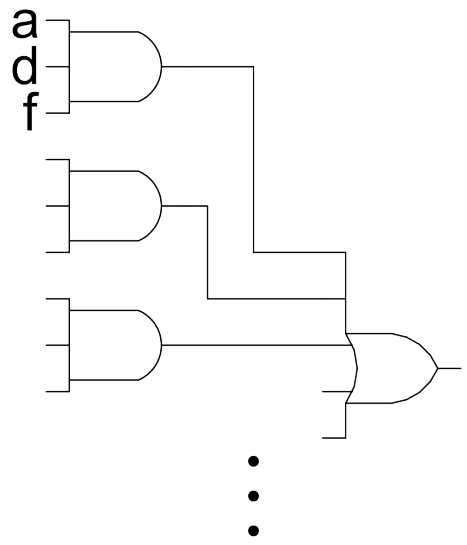
- Example: reduced sum-of-products form

  `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

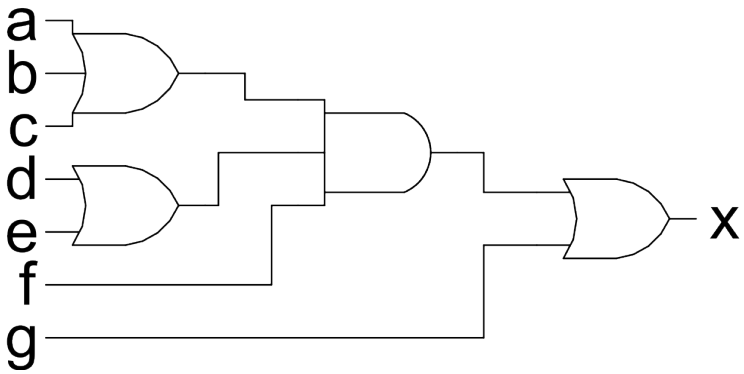


- Factored form:

  `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

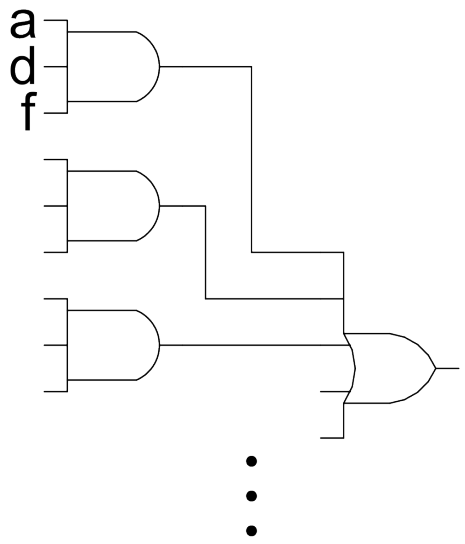
- Example: reduced sum-of-products form

  `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

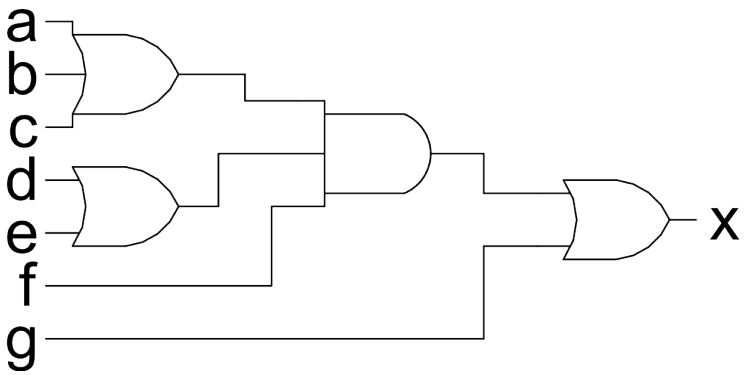


- Factored form:

  `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



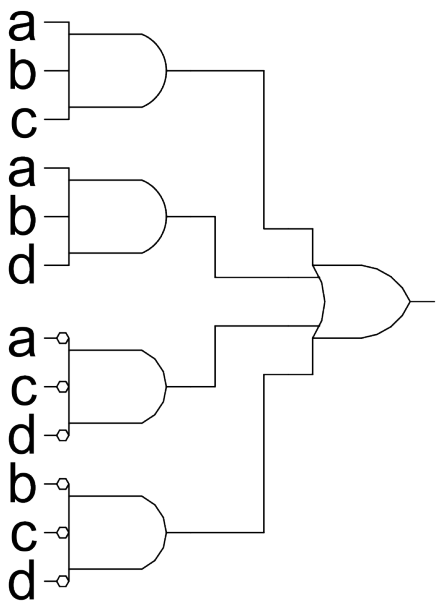
Footnote: NAND would be used in place of all ANDs and ORs.

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.

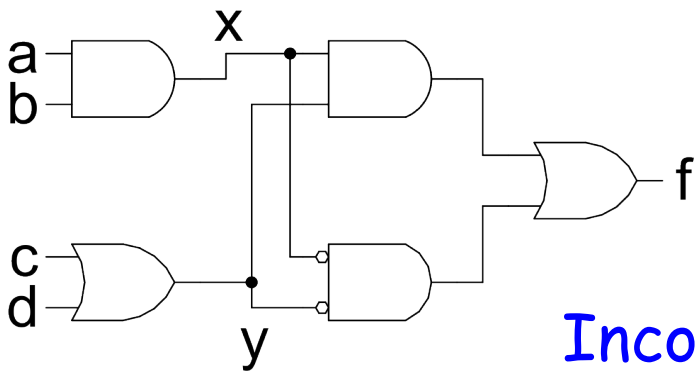
Multi-level Combinational Logic

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



let `x = ab  y = c+d`

`f = xy + x'y'`



Incorporates fanout.

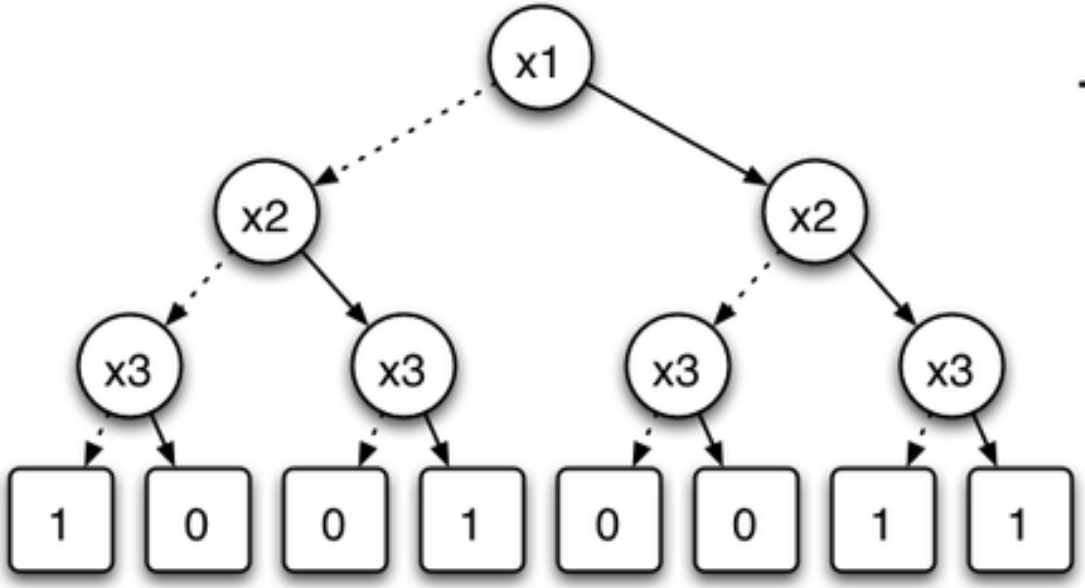
No convenient hand methods exist for multi-level logic simplification:

a) CAD Tools use sophisticated algorithms and heuristics

Guess what? These problems tend to be NP-complete

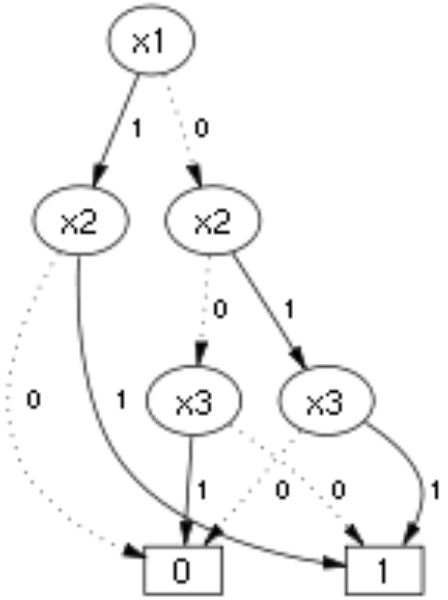
b) Humans and tools often exploit some special structure (example adder)

Binary decision diagrams as the base



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 tree



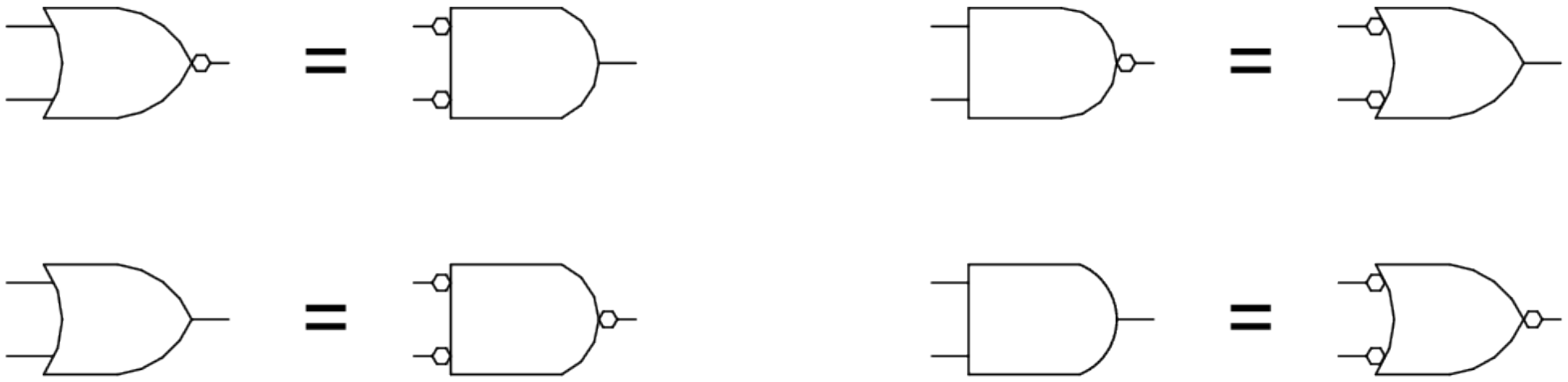
Binary decision diagram (BDD)

NAND-NAND & NOR-NOR Networks

DeMorgan's Law Review:

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

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

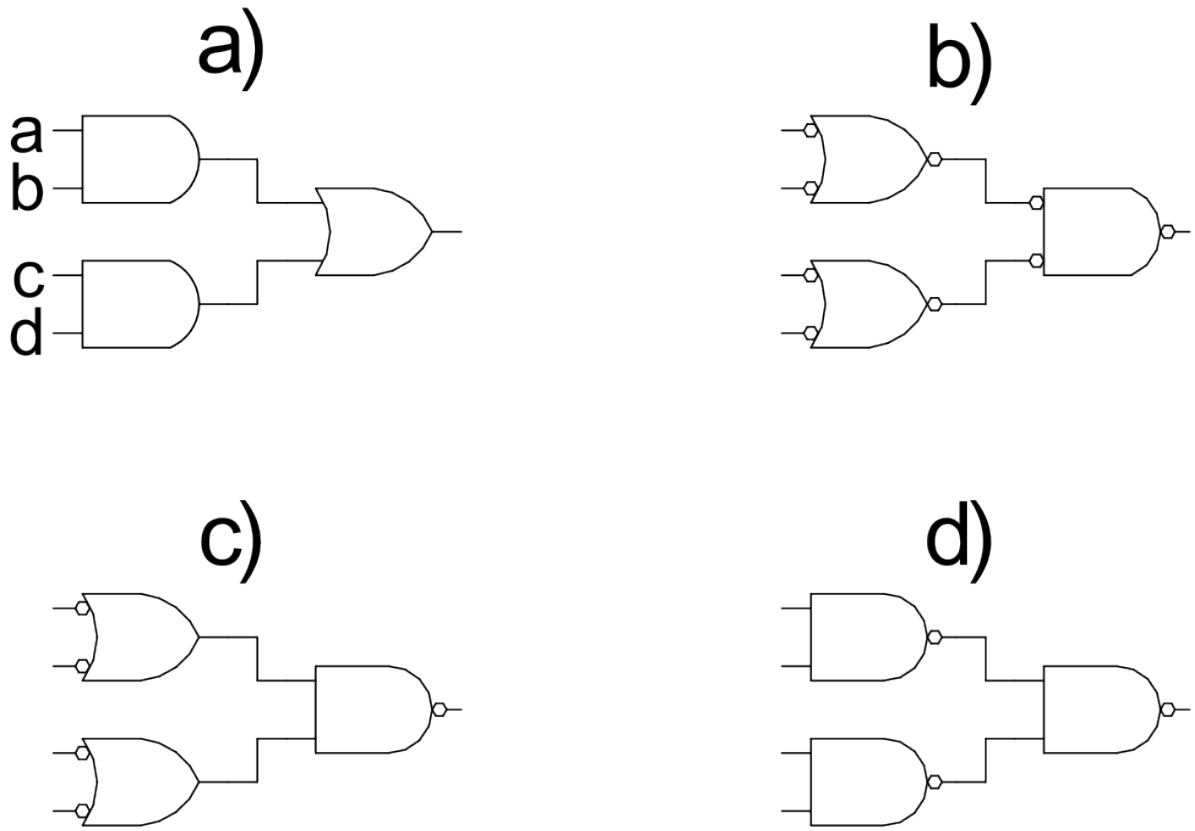


push bubbles or *introduce in pairs* or *remove pairs:*

$(x')' = x$.

NAND-NAND & NOR-NOR Networks

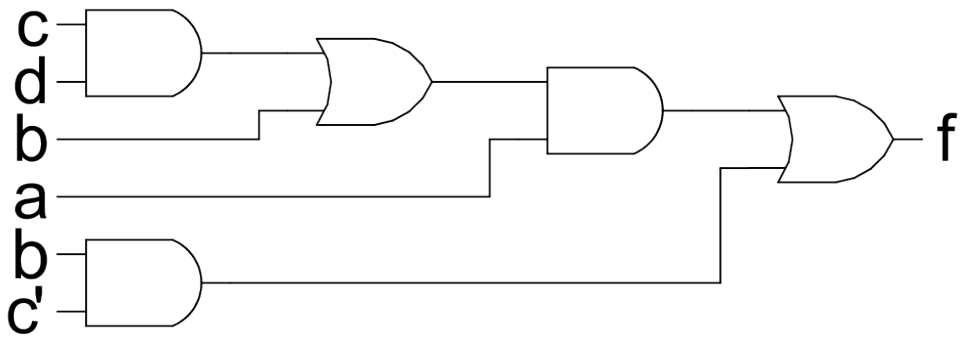
- Mapping from AND/OR to NAND/NAND

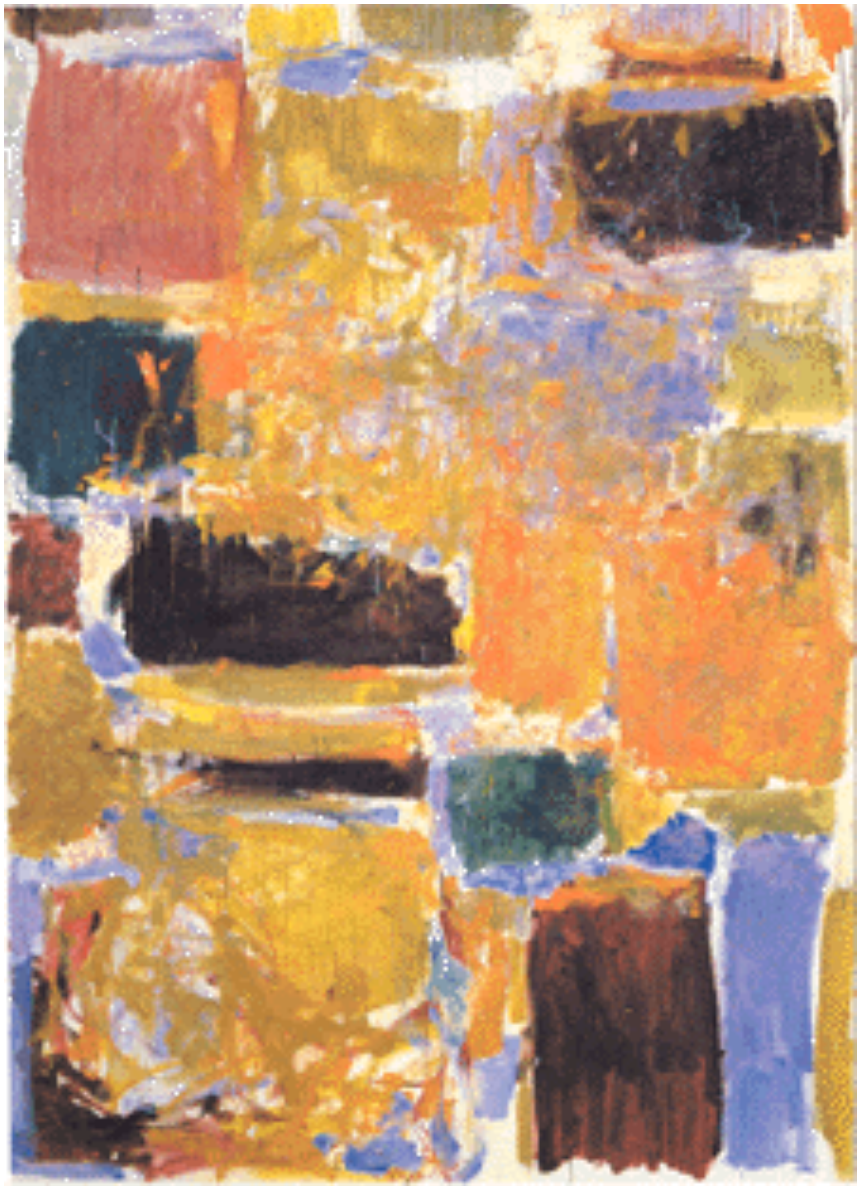


Multi-level Networks

Convert to NANDs:

F = a(b + cd) + bc'

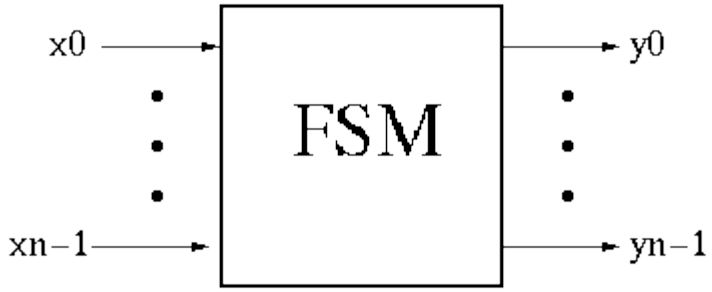




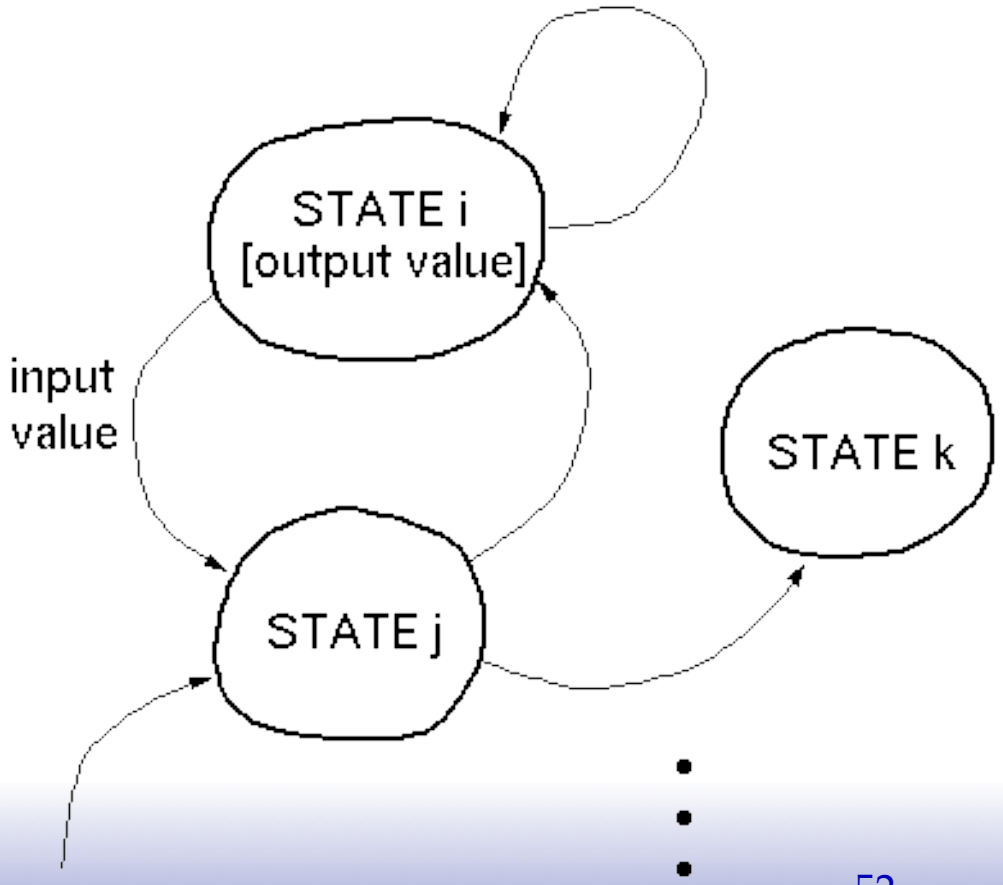
Finite State Machines

Finite State Machines (FSMs)

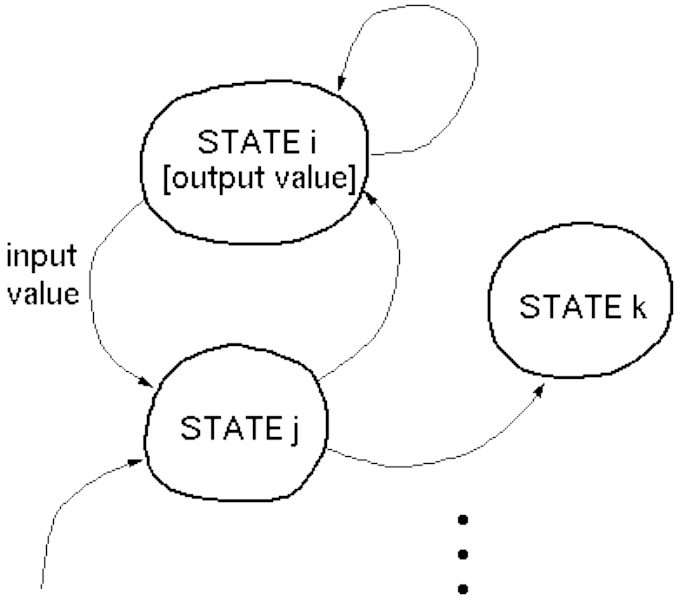
- **FSM** circuits are a type of *sequential circuit*:
  - output depends on present *and* past inputs
    - effect of past inputs is represented by the current *state*

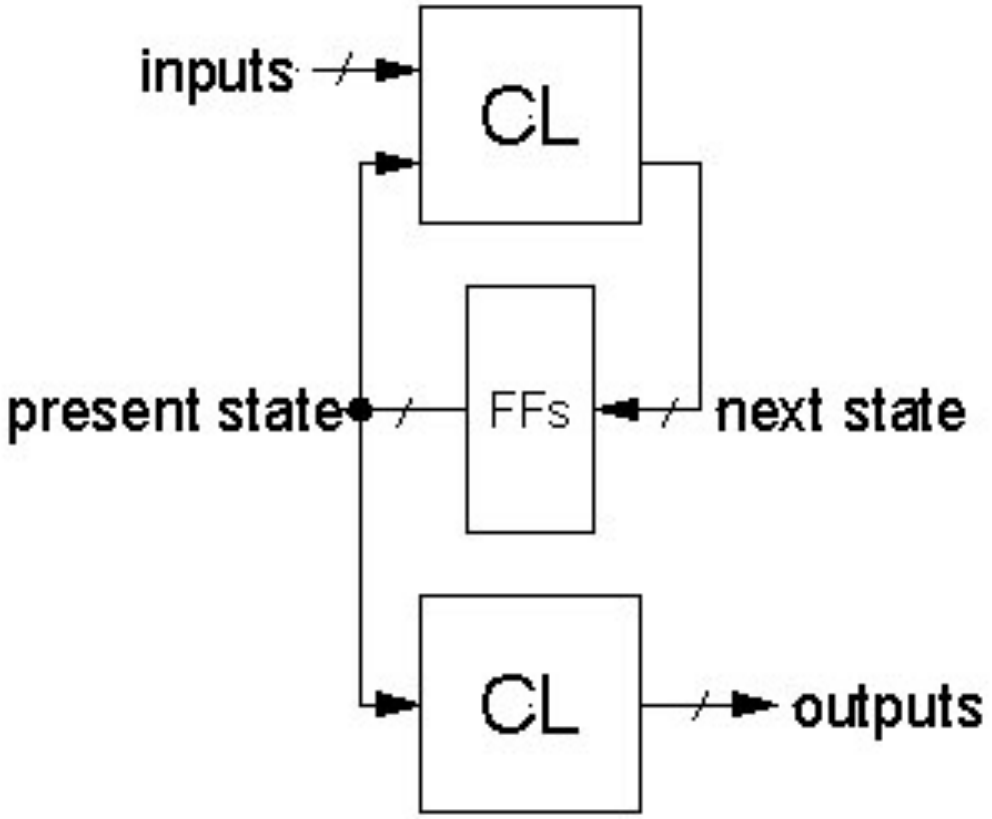


- Behavior is represented by ***State Transition Diagram***:
  - traverse one edge per clock cycle.



FSM Implementation





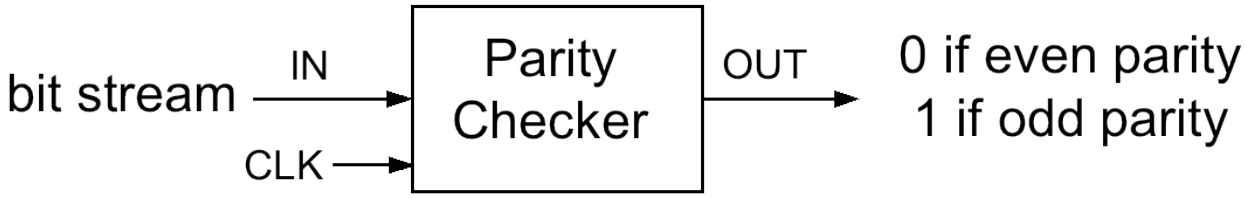
- Flip-flops form *state register*
- number of states ≤ $2^{\text{number of flip-flops}}$
- CL (combinational logic) calculates next state and output
- Remember: The FSM follows exactly one edge per cycle.

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

Parity Checker Example

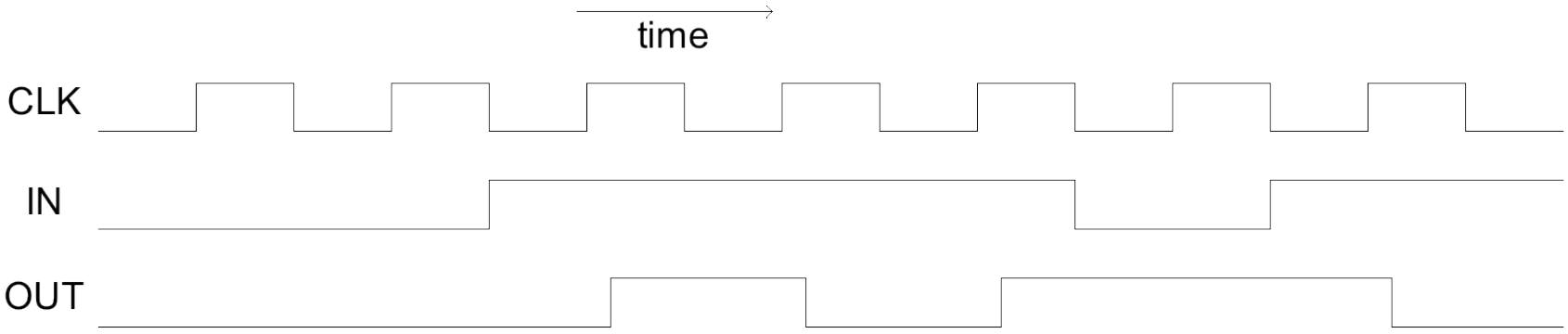
A string of bits has "even parity" if the number of 1's in the string is even.

- 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

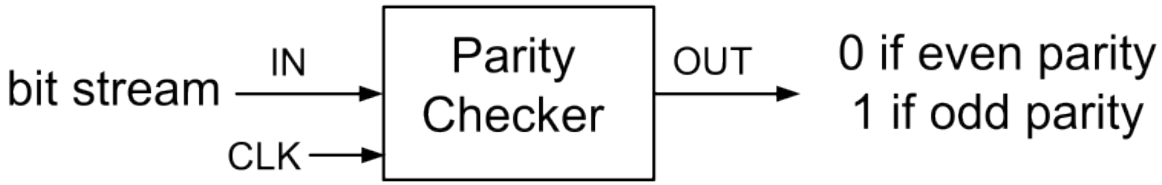
even even odd even odd odd even



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)



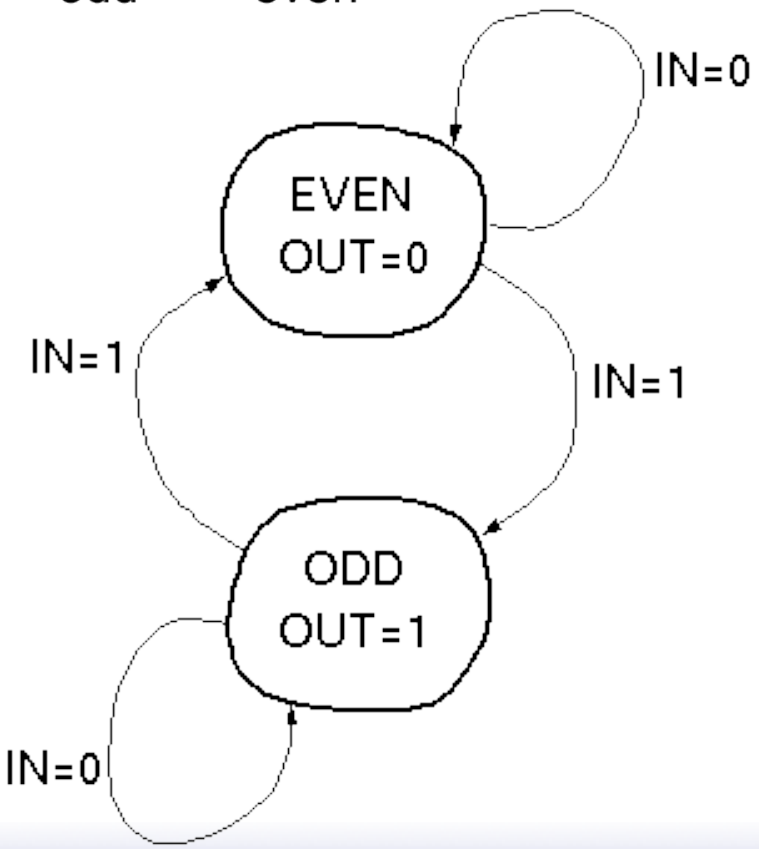
example: 0 0 1 1 1 0 1

even even odd even odd odd even

time

"State Transition Diagram"

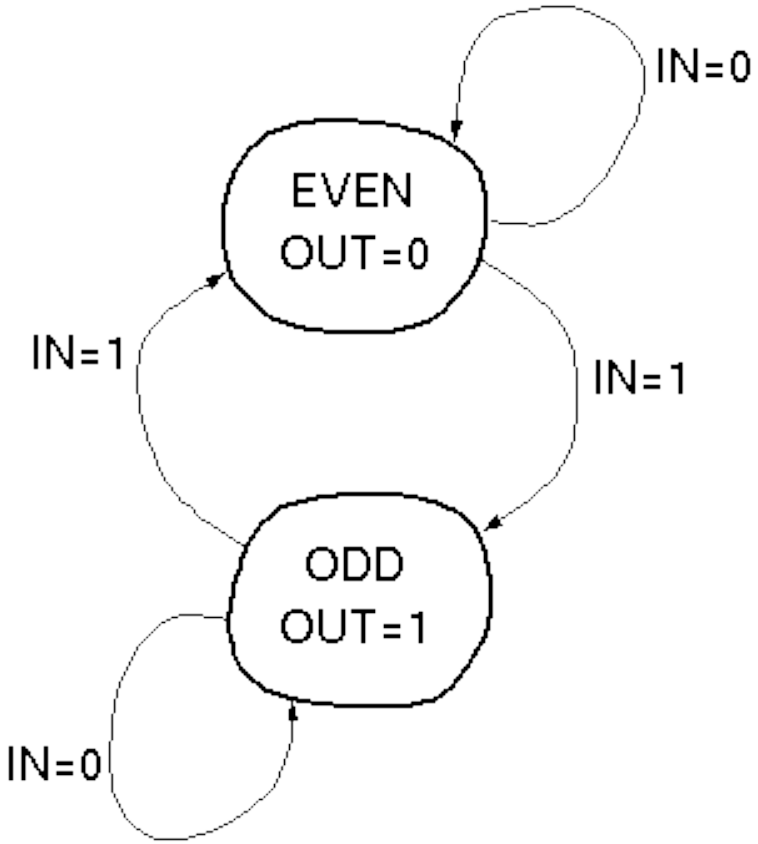
- circuit is in one of two "states".
- transition on each cycle with each new input, over exactly one arc (edge).
- Output depends on which state the circuit is in.



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

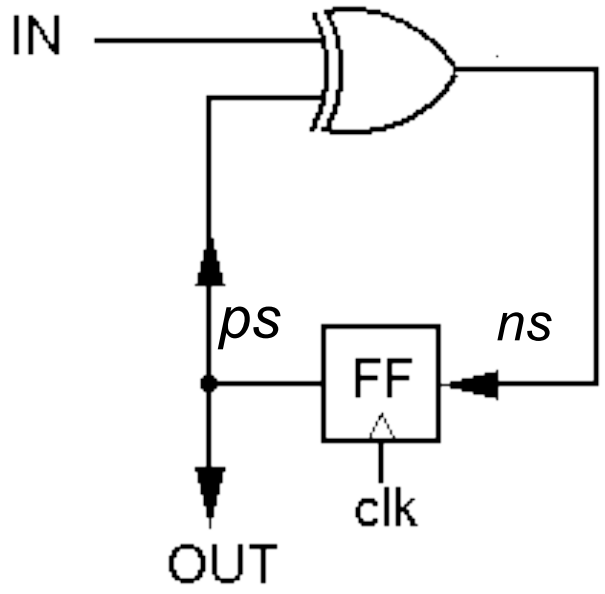
NS = PS xor IN

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.

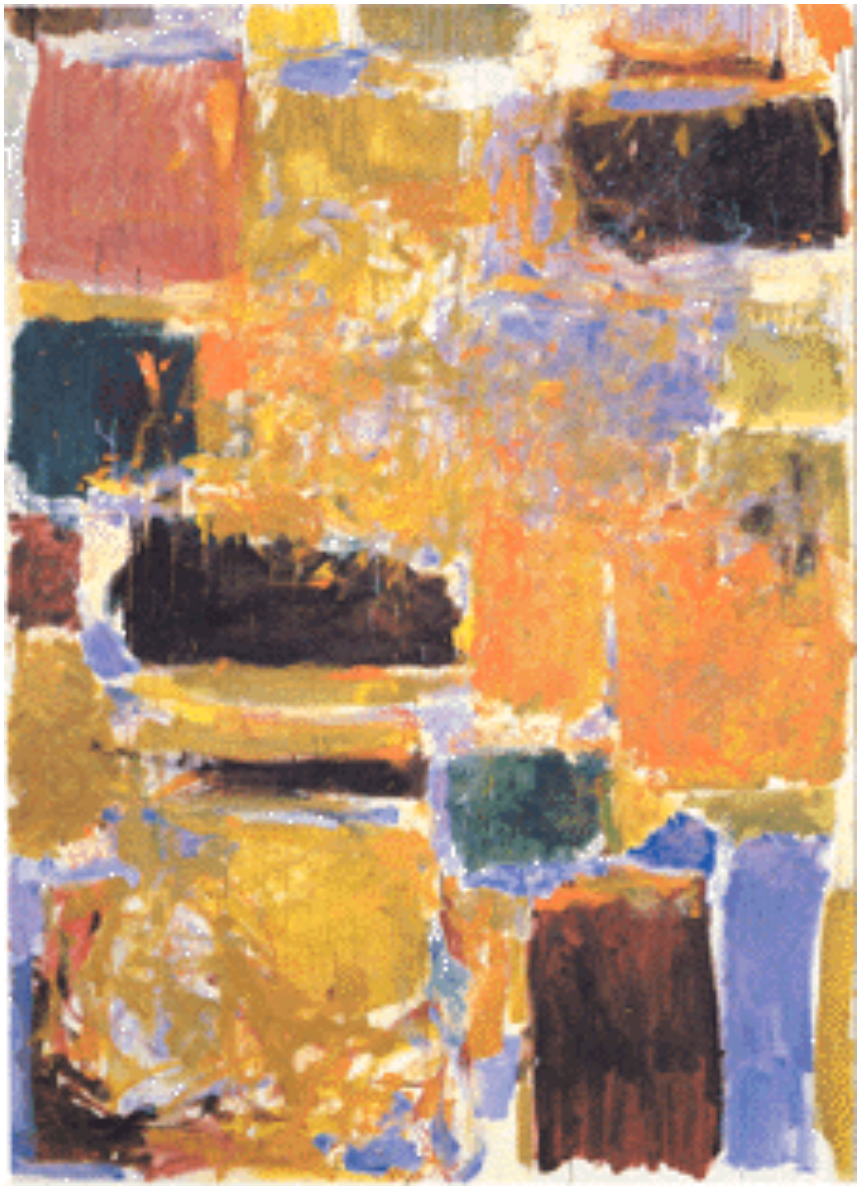
Formal Design Process

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***

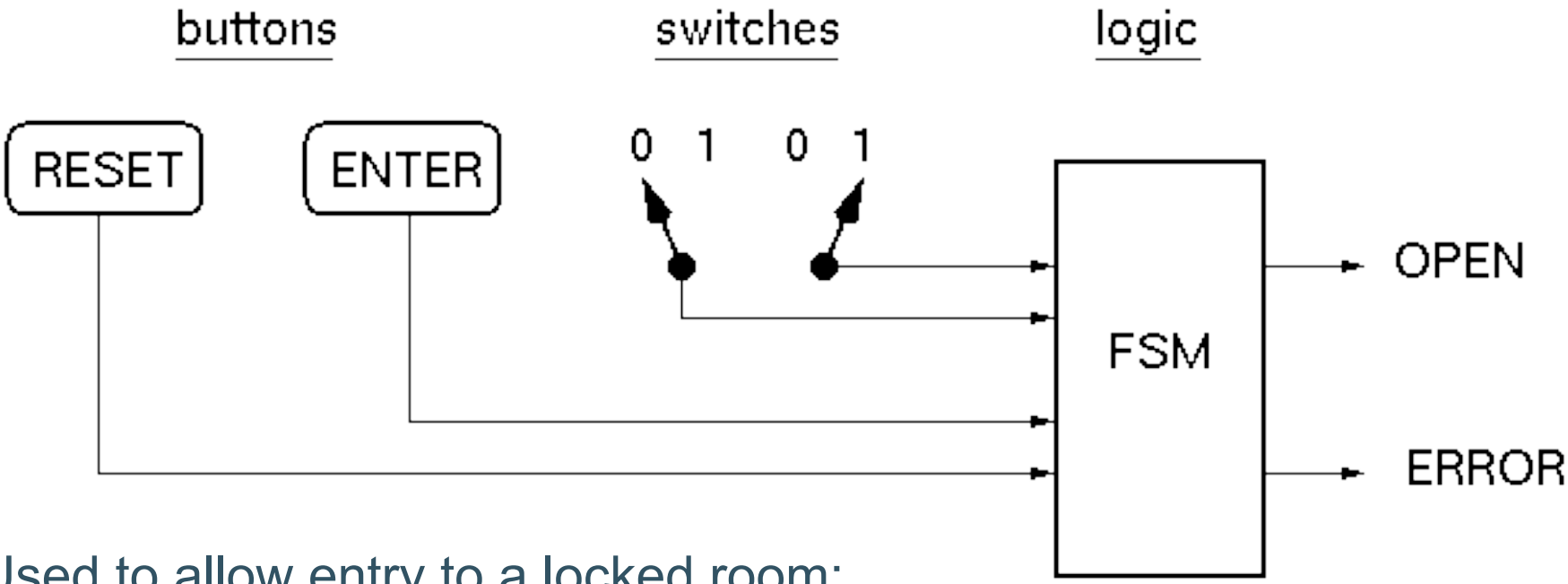
   Register to hold state

   Combinational Logic for Next State and Outputs



FSM Design Example

Combination Lock Example



- Used to allow entry to a locked room:

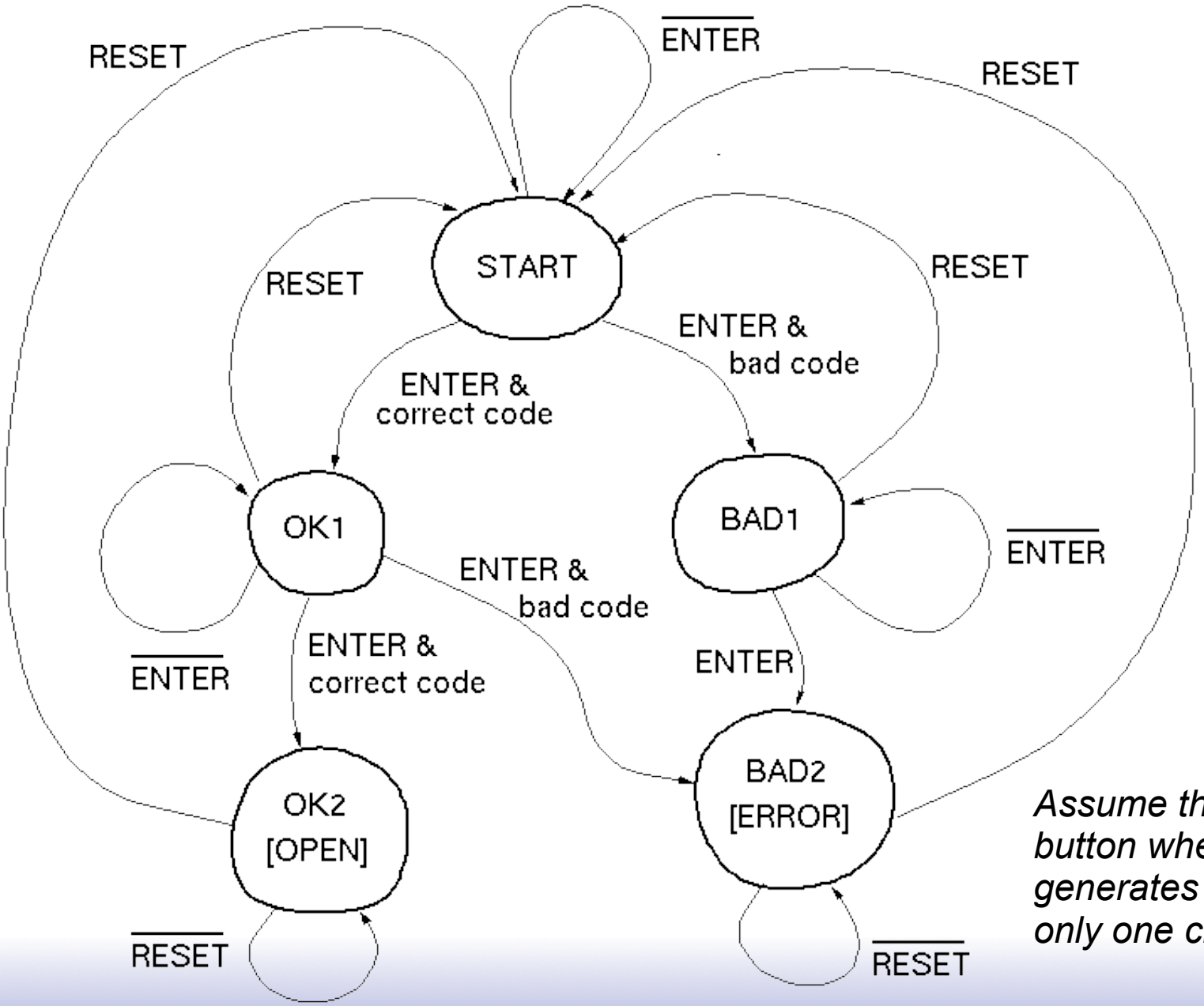
  2-bit serial combination. Example 01,11:

  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

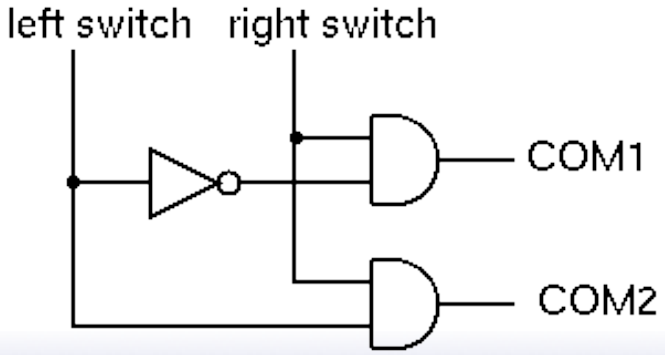


Assume the ENTER button when pressed generates a pulse for only one clock cycle.

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

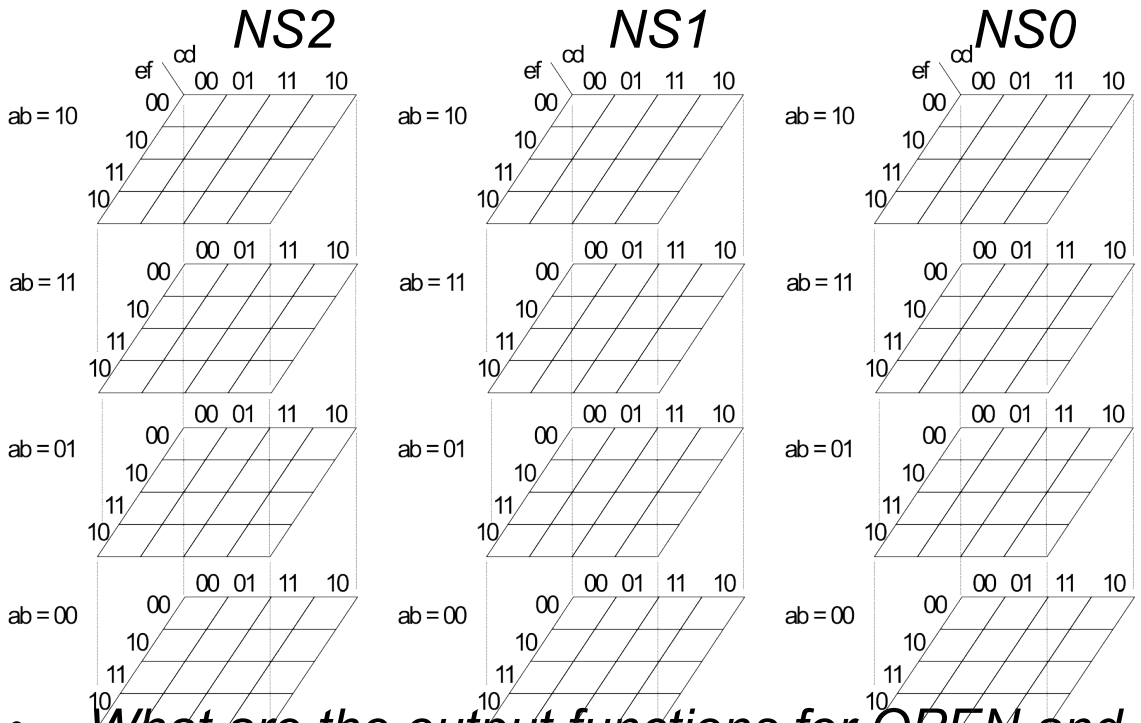
ENTER	COM1	COM2	PS2	PS1	PS0	NS2	NS1	NS0
0	0	0	0	0	0	0	0	0
0	0	1	0	0	0	0	0	0
0	1	0	0	0	0	0	0	0
0	1	1	0	0	0	0	0	0
1	0	0	0	0	0	1	0	0
1	0	1	0	0	0	1	0	0
1	1	0	0	0	0	0	0	1
1	1	1	0	0	0	0	0	1
0	0	0	0	0	1	0	0	1
0	0	1	0	0	1	0	0	1
0	1	0	0	0	1	0	0	1
0	1	1	0	0	1	0	0	1
1	0	0	0	0	1	1	0	1
1	1	0	0	0	1	1	0	1
1	0	1	0	0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

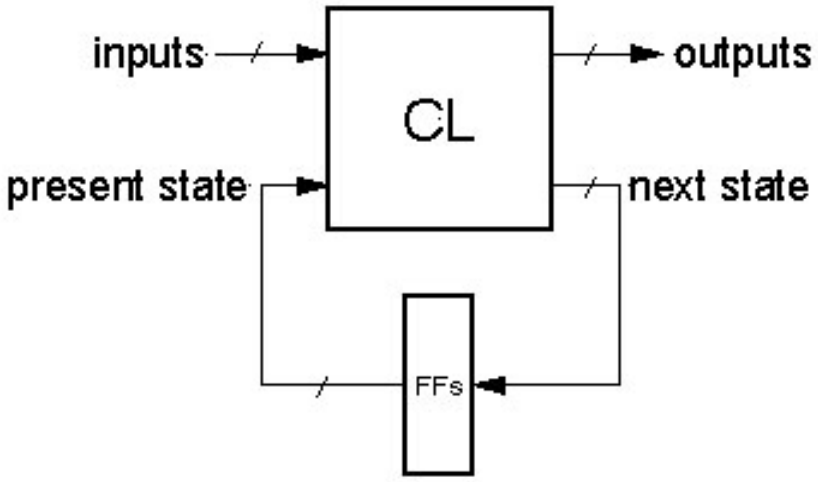
BAD1=100, BAD2=101

- *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 = $2^{\#\text{ of Flip-flops}}$

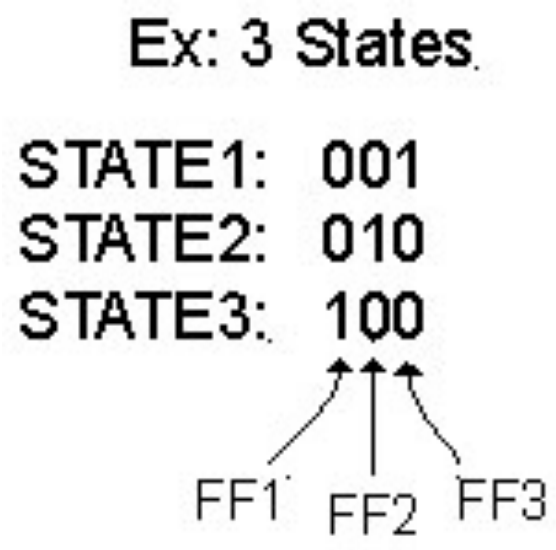
  Example:

  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

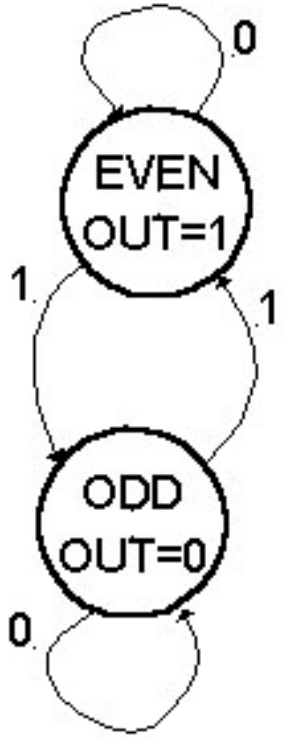
- **One-hot encoding of states.**
- One FF per state.

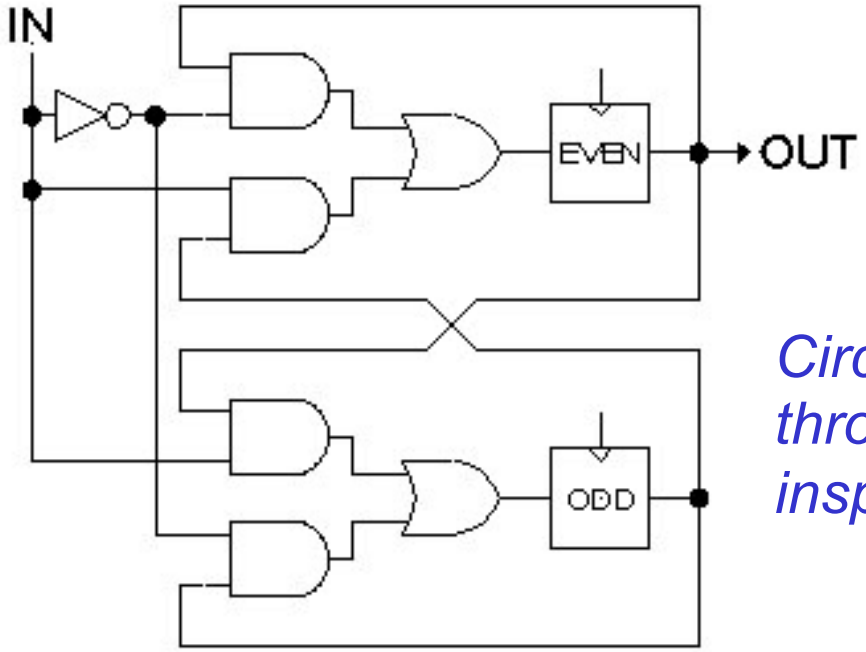


- Why one-hot encoding?
  - Simple design procedure.
    - Circuit matches state transition diagram (example next page).
  - Often can lead to simpler and faster “next state” and output logic.
- Why not do this?
  - Can be costly in terms of Flip-flops for FSMs with large number of states.
- FPGAs are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

One-hot encoded FSM

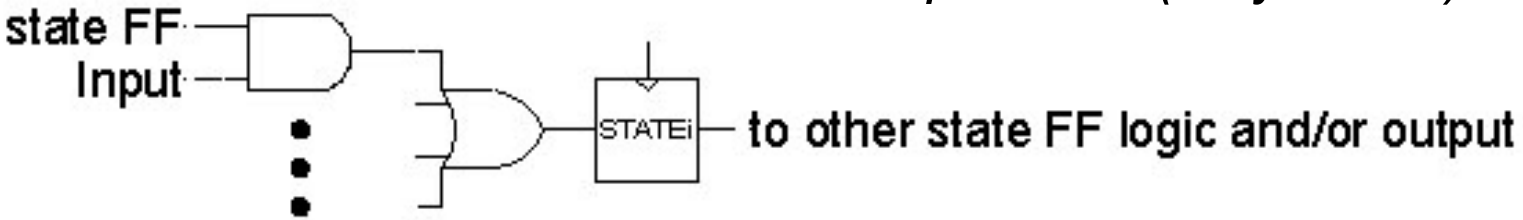
- Even Parity Checker Circuit:





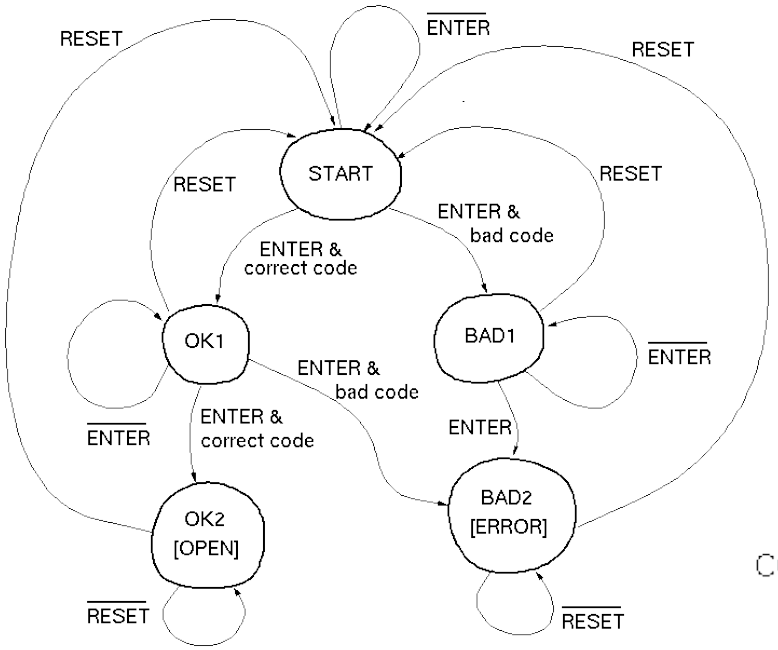
Circuit generated through direct inspection of the STD.

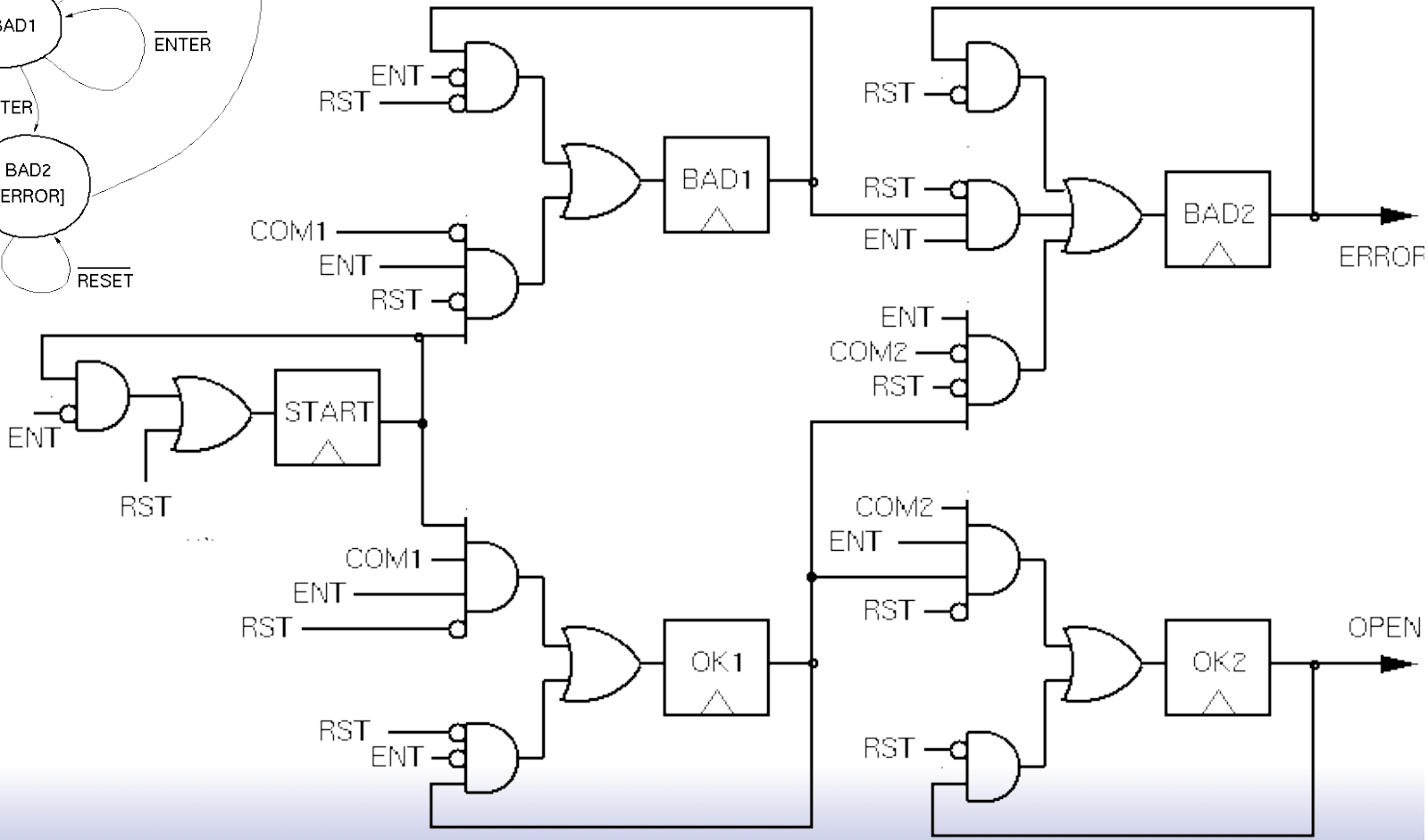
- In General:

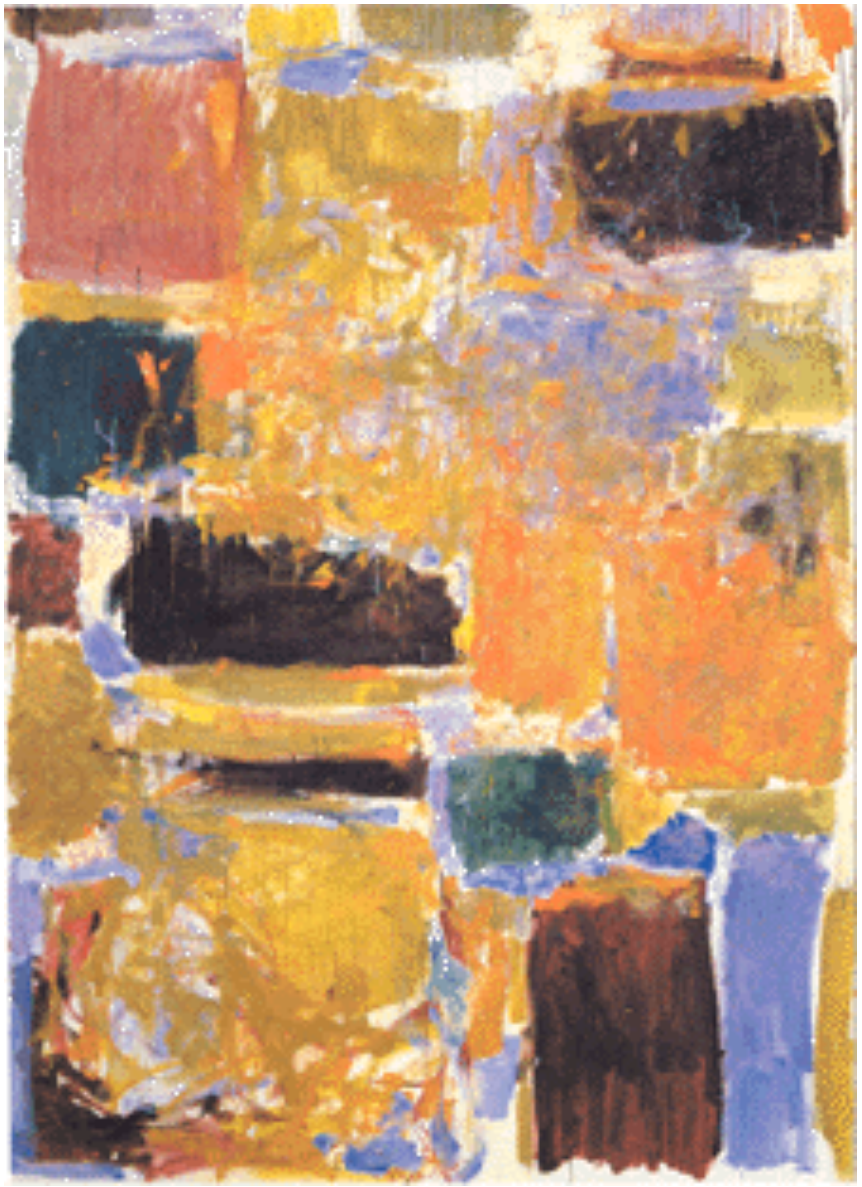


- *FFs must be initialized for correct operation (only one 1)*

One-hot encoded combination lock



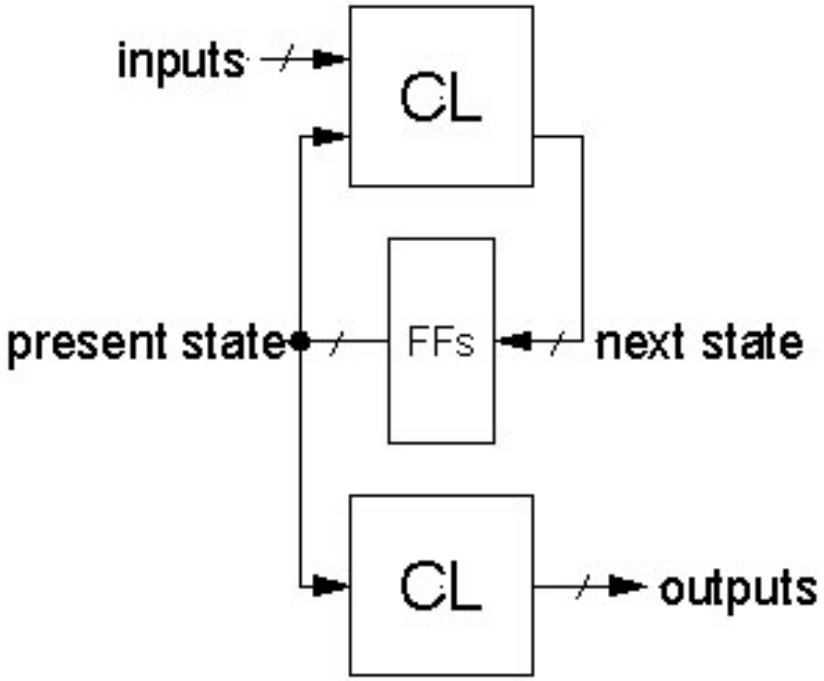




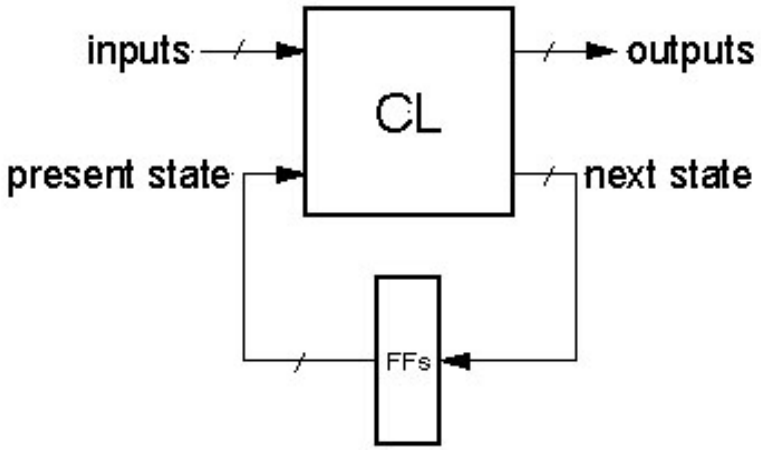
Moore Versus Mealy Machines

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*”:

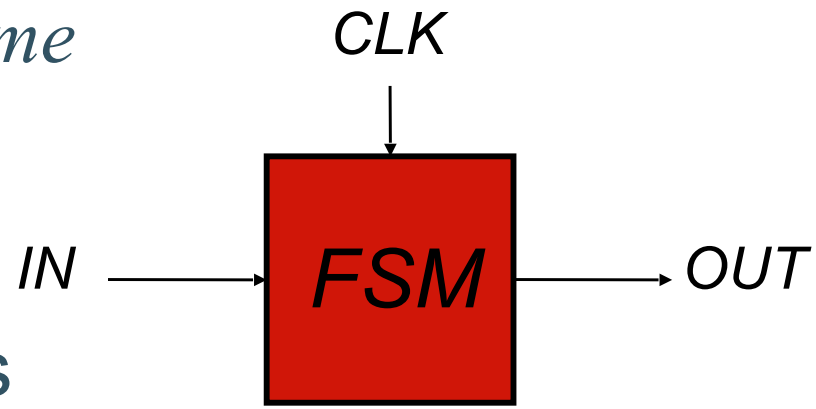


Finite State Machines

- **Example: Edge Detector**

  Bit are received one at a time (one per cycle),

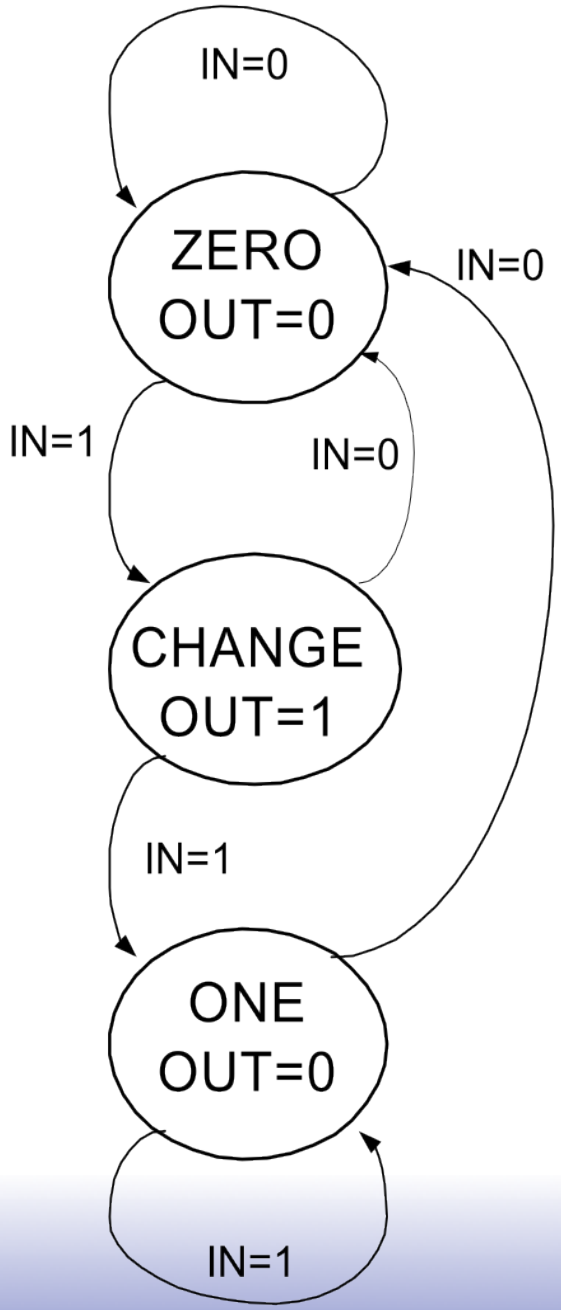
  such as: 000111010 ⟶ *time*



  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.

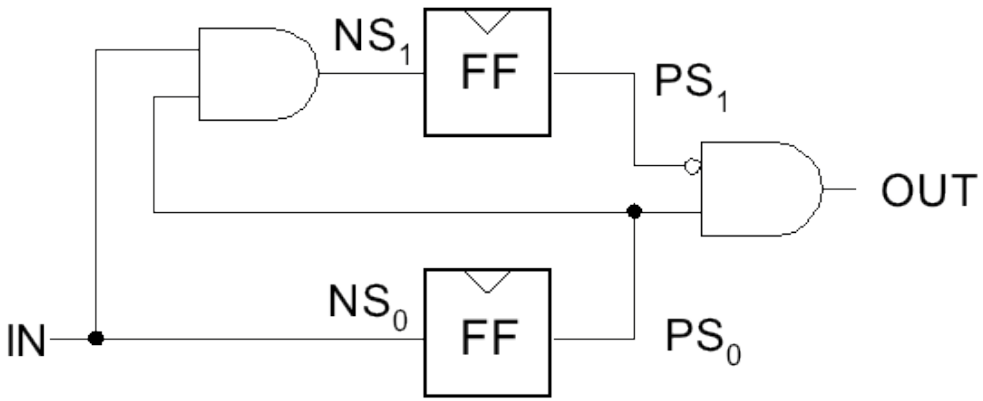
State Transition Diagram Solution A



	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



PS

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

$NS_1 = IN\ PS_0$

PS

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

$NS_0 = IN$

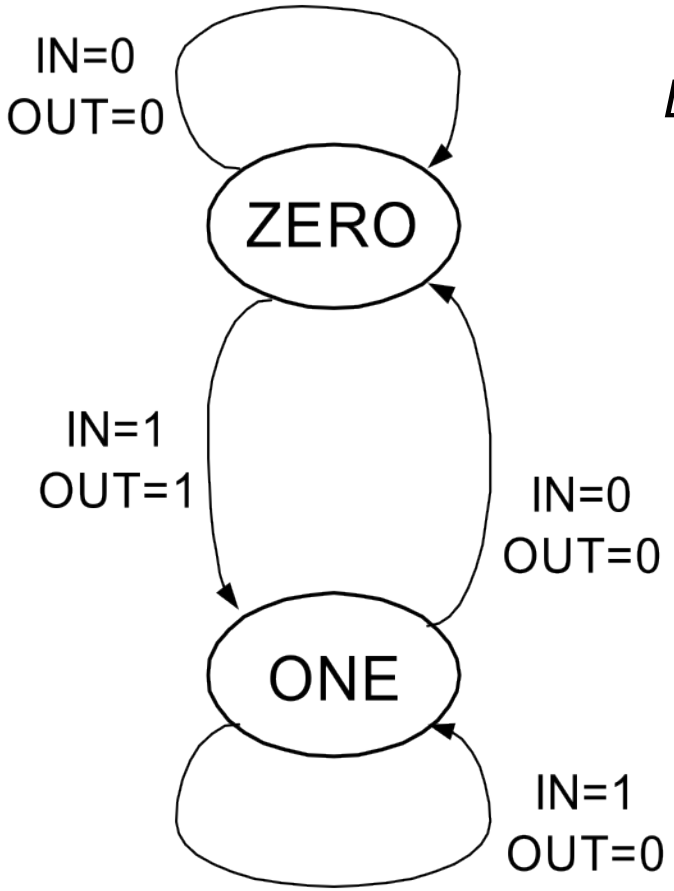
PS

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

$OUT = \overline{PS_1}\ PS_0$

Solution B

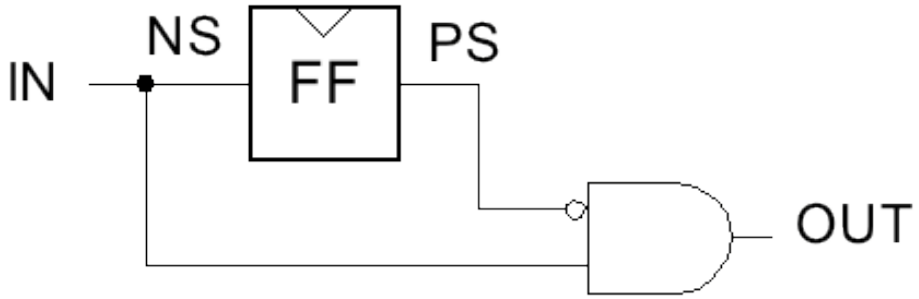
Output depends not only on PS but also on input, IN



Let ZERO=0, ONE=1

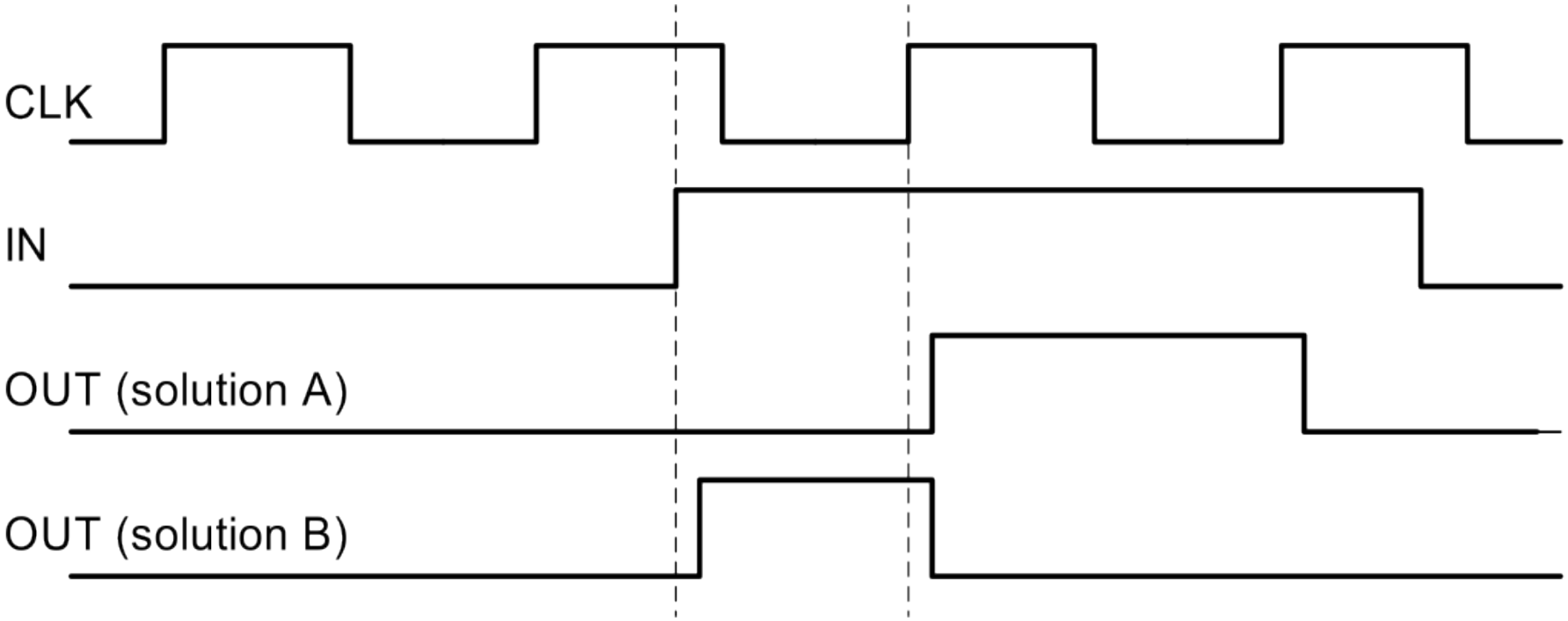
IN	PS	NS	OUT
0	0	0	0
0	1	0	0
1	0	1	1
1	1	1	0

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

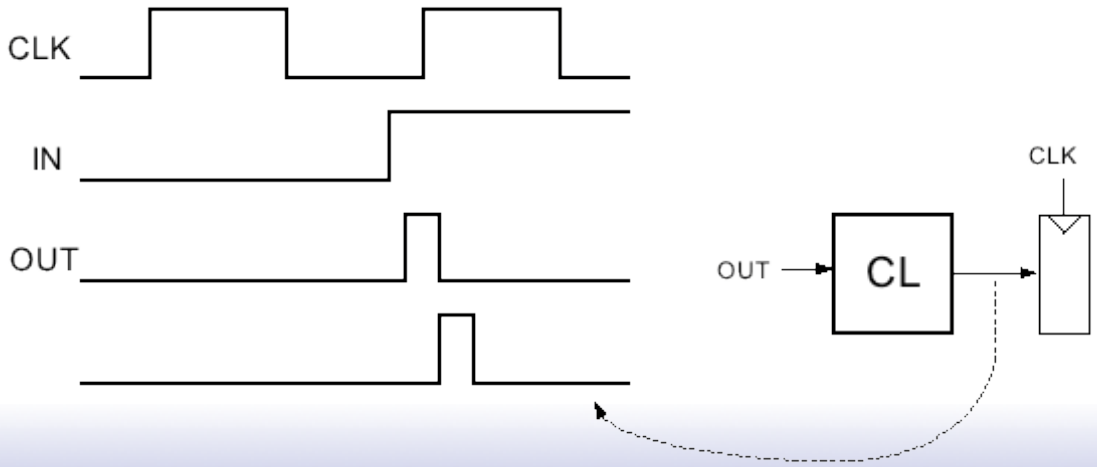
Moore Machine

- output function only of PS
- maybe more states (why?)
- synchronous outputs
  - Input glitches not send at output
  - one cycle “delay”
  - full cycle of stable output

Solution B

Mealy Machine

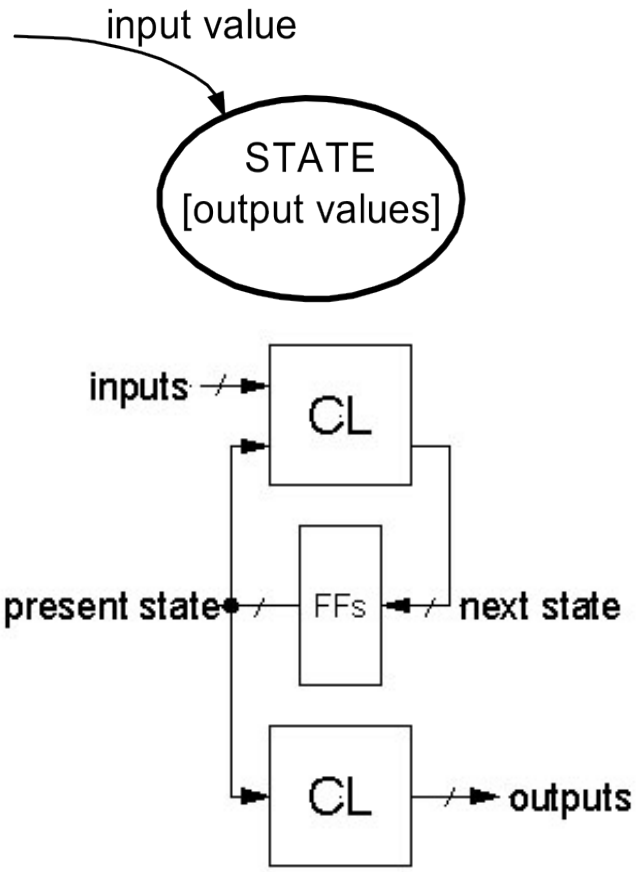
- *output function of both PS & input*
- *maybe fewer states*
- *asynchronous outputs*
  - *if input glitches, so does output*
  - *output immediately available*
  - *output may not be stable long enough to be useful (below):*



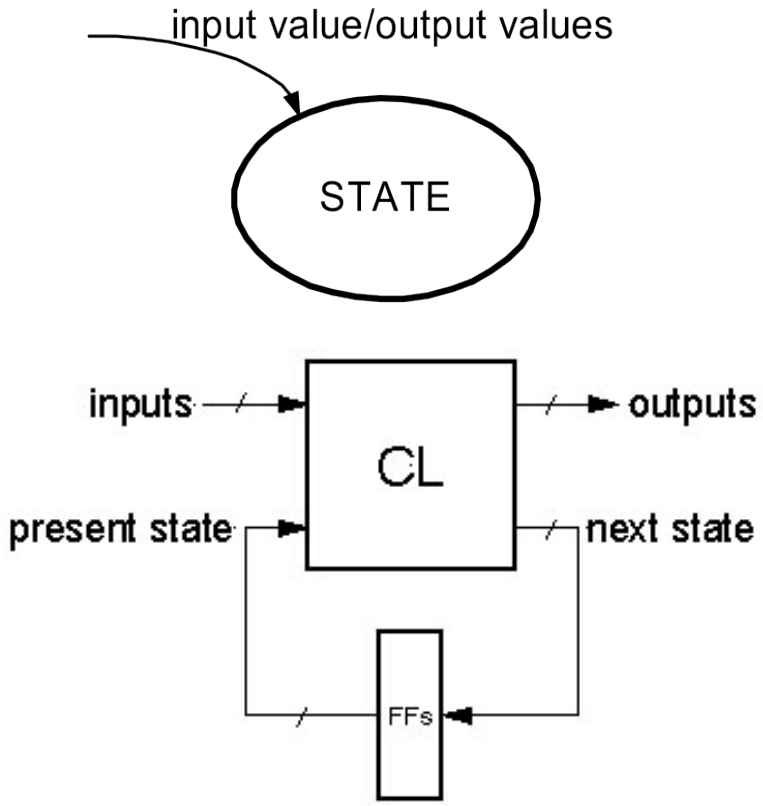
If output of Mealy FSM goes through combinational logic before being registered, the CL might delay the signal and it could be missed by the clock edge.

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: