EECS 151/251A
Spring 2023
Digital Design and Integrated Circuits

Instructor:
Wawrzynek

Lecture 14 - Exam1 Review

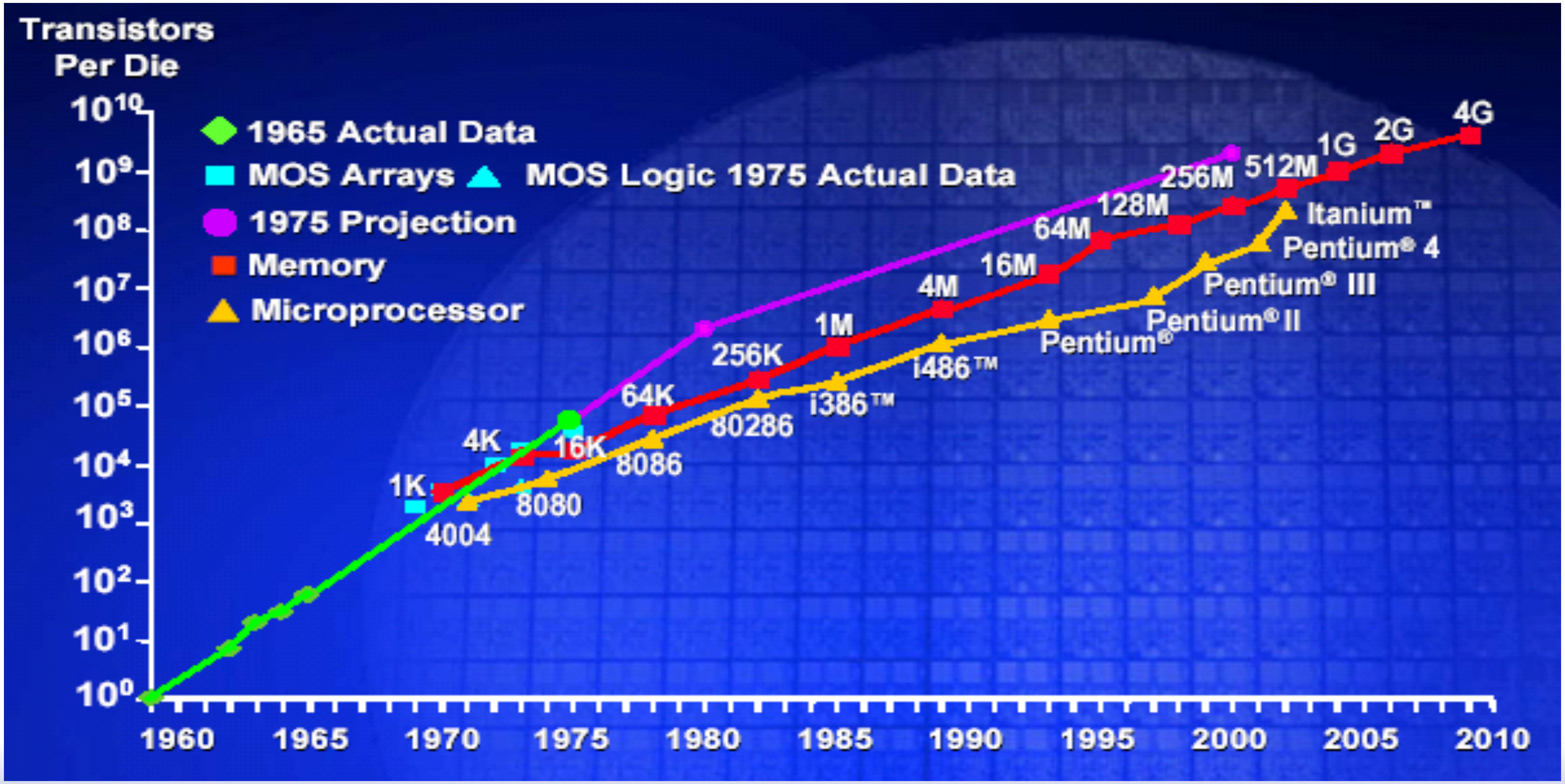
Announcements

- ***No class Thursday 3/9.***
- ***Midterm Exam 6-9PM***
  - ***Latimer 120 (alternate seating)***
  - ***Exam covers Lectures 1 - 12 and HW 1 - 6***
  - ***One double sided handwritten sheet of paper allowed. No calculators.***
- Homework #6 assignment solutions posted Monday 3/6 - part of exam 1.
- No homework posted Friday 3/3 nor due Monday 3/13.
- *No Wawrzynek office hour(s) today*

Review with sample slides

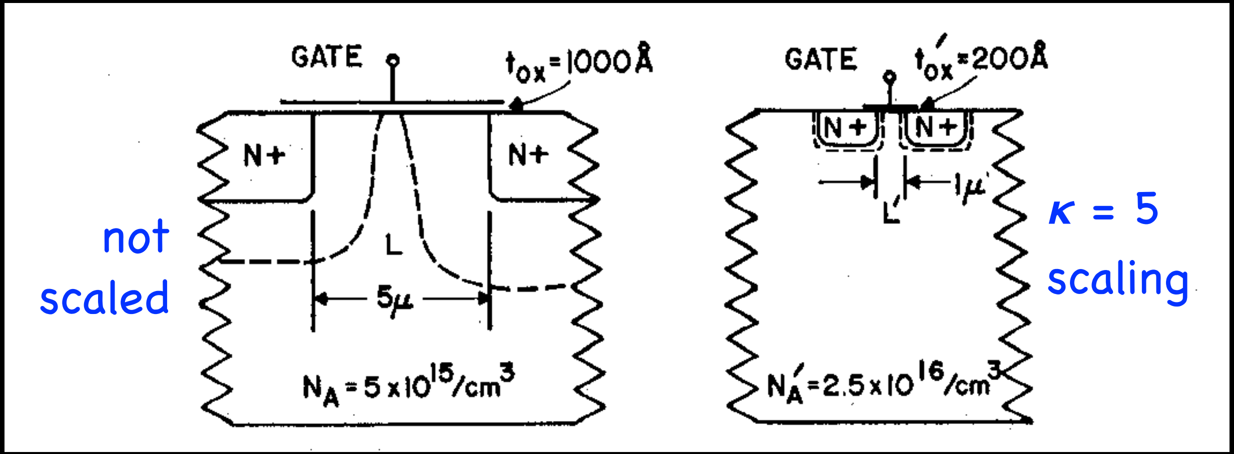
- Do not study only the following slides. These are just representative of what you need to know.
- Go back and study the entire lecture.

Moore's Law – 2x transistors per 1-2 yr



Dennard Scaling

Things we do: scale dimensions, doping, Vdd.



not scaled

κ = 5 scaling

What we get:

κ^2 as many transistors at the same power density!

Whose gates switch κ times faster!

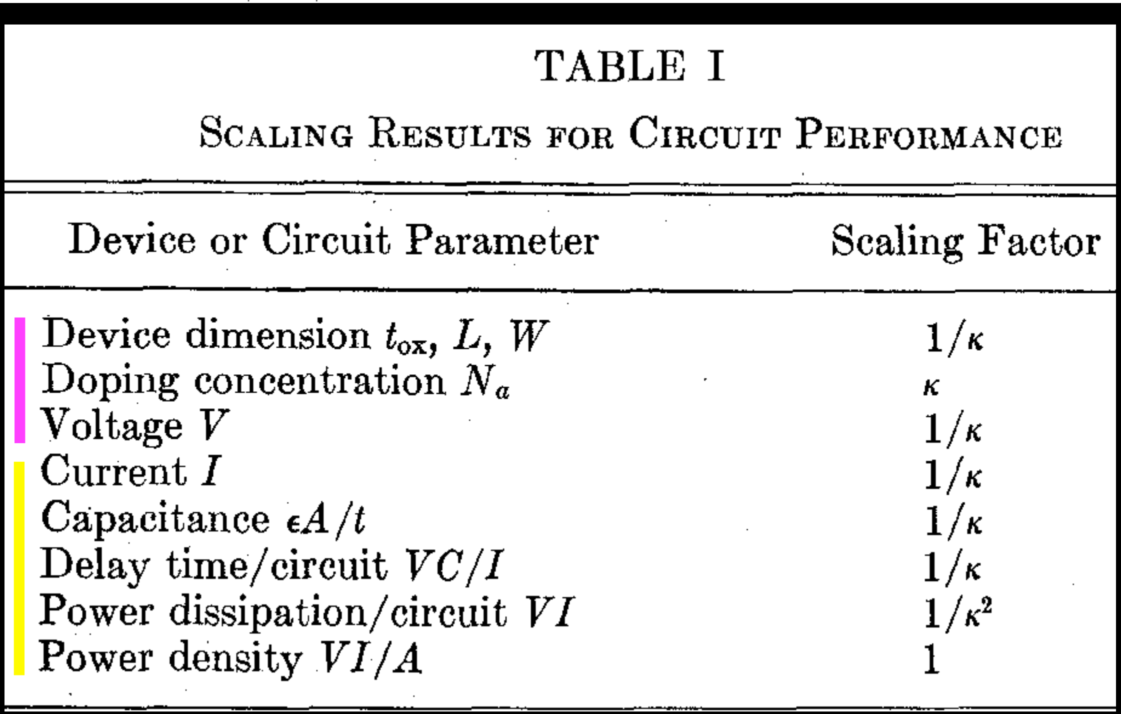


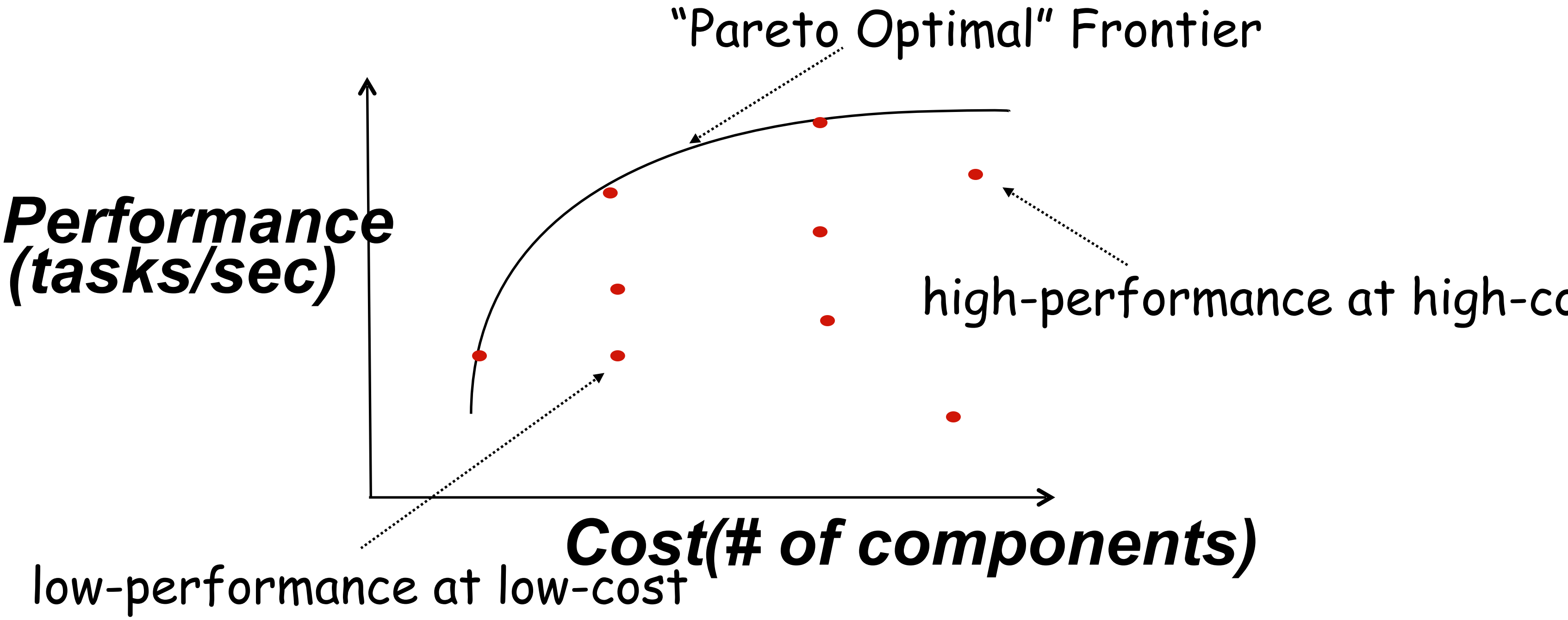
TABLE I

SCALING RESULTS FOR CIRCUIT PERFORMANCE

Device or Circuit Parameter	Scaling Factor
Device dimension t_{ox}, L, W	$1/\kappa$
Doping concentration N_a	κ
Voltage V	$1/\kappa$
Current I	$1/\kappa$
Capacitance $\epsilon A/t$	$1/\kappa$
Delay time/circuit VC/I	$1/\kappa$
Power dissipation/circuit VI	$1/\kappa^2$
Power density VI/A	1

Power density scaling ended in 2003 (Pentium 4: 3.2GHz, 82W, 55M FETs).

Design Space & Optimality



"Pareto Optimal" Frontier

Performance (tasks/sec)

high-performance at high-co

Cost(# of components)

low-performance at low-cost

Cost

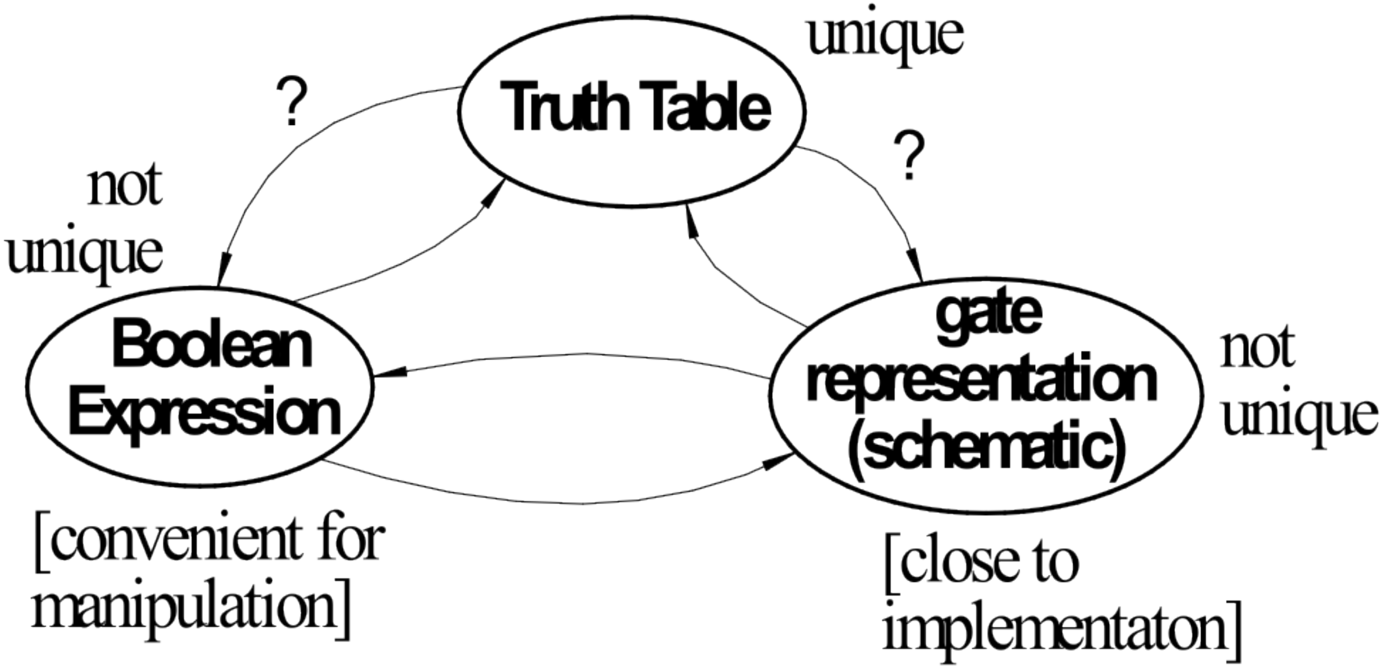
- **Non-recurring** engineering (NRE) costs
- Cost to develop a design (product)
  - Amortized over all units shipped
  - E.g. $20M in development adds $.20 to each of 100M units
- **Recurring** costs
  - Cost to manufacture, test and package a unit
  - Processed wafer cost is ~10k (around 16nm node) which yields:
    - 50-100 large FPGAs or GPUs
    - 200 laptop CPUs
    - >1000 cell phone SoCs

$$\text{cost per IC} = \text{variable cost per IC} + \frac{\text{fixed cost}}{\text{volume}}$$

$$\text{variable cost} = \frac{\text{cost of die} + \text{cost of die test} + \text{cost of packaging}}{\text{final test yield}}$$

Relationship Among Representations

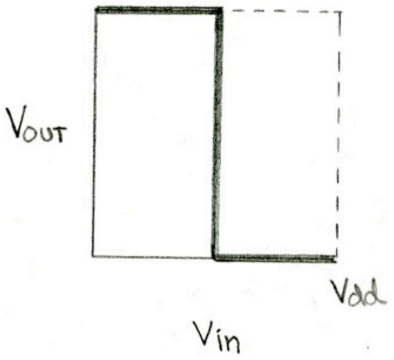
- Theorem: Any Boolean function that can be expressed as a truth table can be written as an expression in Boolean Algebra using AND, OR, NOT.

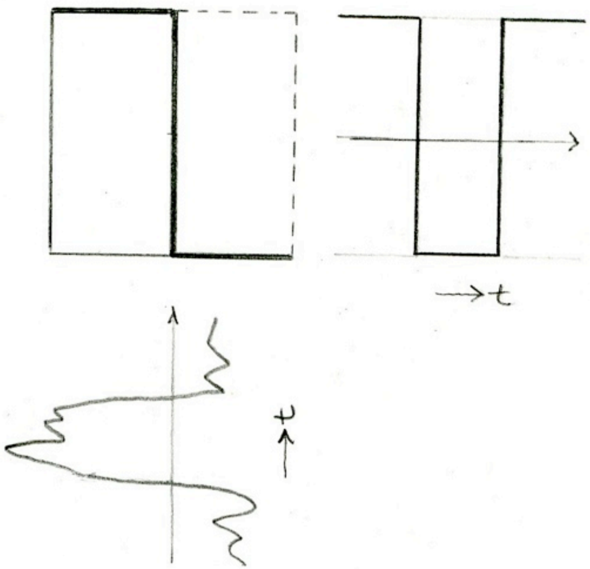


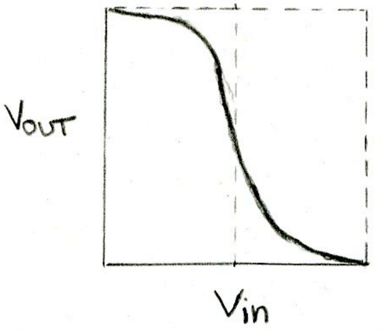
How do we convert from one to the other?

Inverter Example of Restoration

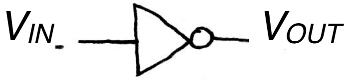
Example (look at 1-input gate, to keep it simple):







Idealize Inverter



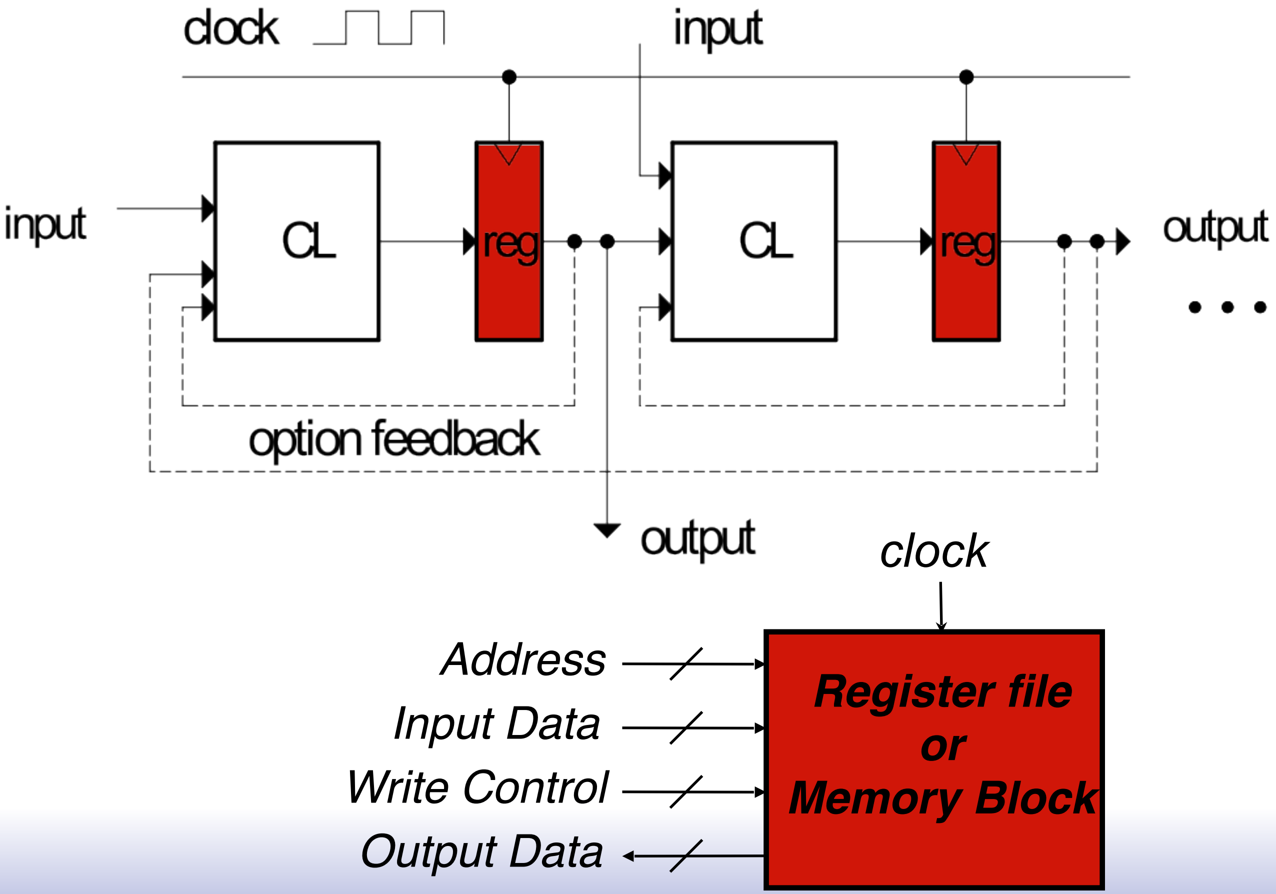
Actual Inverter

- Inverter acts like a “non-linear” amplifier
- The non-linearity is critical to restoration
- Other logic gates act similarly with respect to input/output relationship.

Register Transfer Level Abstraction (RTL)

Any synchronous digital circuit can be represented with:

- Combinational Logic Blocks (CL), plus
- State Elements (registers or memories)



- State elements are mixed in with CL blocks to control the flow of data.
- Sometimes used in large groups by themselves for “long-term” data storage.

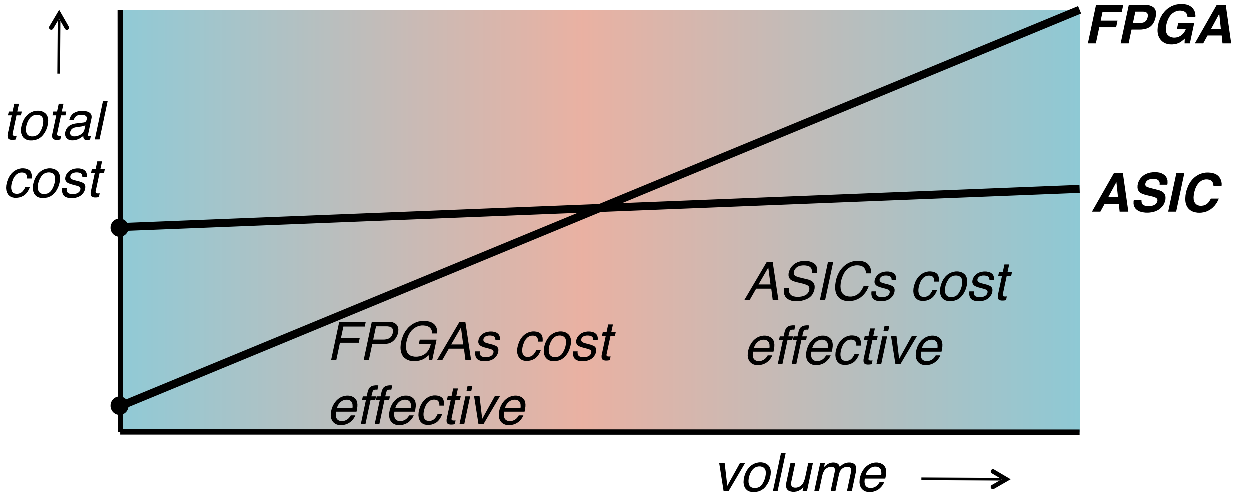
Implementation Alternative Summary

Full-custom:	All circuits/transistors layouts optimized for application.
Standard-cell:	Small function blocks/"cells" (gates, FFs) automatically placed and routed.
Gate-array (structured ASIC):	Partially prefabricated wafers with arrays of transistors customized with metal layers or vias.
FPGA:	Prefabricated chips customized with loadable latches or fuses.
Microprocessor:	Instruction set interpreter customized through software.
Domain Specific Processor:	Special instruction set interpreters (ex: DSP, NP, GPU).

These days, "ASIC" almost always means Standard-cell.

What are the important metrics of comparison?

FPGA versus ASIC



- **ASIC:** Higher NRE costs (10's of $M). Relatively Low cost per die (10's of $ or less).
- **FPGAs:** Low NRE costs. Relatively low silicon efficiency ⇒ high cost per part (> 10's of $ to 1000's of $).
- **Cross-over volume** from cost effective FPGA design to ASIC was often in the 100K range.

Hardware Description Languages

- Basic Idea:
  - Language constructs describe circuits with two basic forms:
  - **Structural descriptions:** connections of components. Nearly one-to-one correspondence to with schematic diagram.
  - **Behavioral descriptions:** use high-level constructs (similar to conventional programming) to describe the circuit function.
- Originally invented for simulation.
  - "logic synthesis" tools exist to automatically convert to gate level representation.
  - High-level constructs greatly improves designer productivity.
  - However, this may lead you to falsely believe that hardware design can be reduced to writing programs*

"Structural" example:

```
Decoder(output x0,x1,x2,x3;
    inputs a,b)
    {
        wire abar, bbar;
        inv(bbar, b);
        inv(abar, a);
        and(x0, abar, bbar);
        and(x1, abar, b   );
        and(x2, a,    bbar);
        and(x3, a,    b   );
    }
```

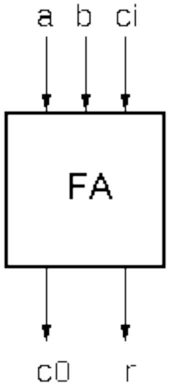
"Behavioral" example:

```
Decoder(output x0,x1,x2,x3;
    inputs a,b)
    {
        case [a b]
           00: [x0 x1 x2 x3] = 0x8;
           01: [x0 x1 x2 x3] = 0x4;
           10: [x0 x1 x2 x3] = 0x2;
           11: [x0 x1 x2 x3] = 0x1;
        endcase;
    }
```

Warning: this is a fake HDL!

*Describing hardware with a language is similar, however, to writing a parallel program.

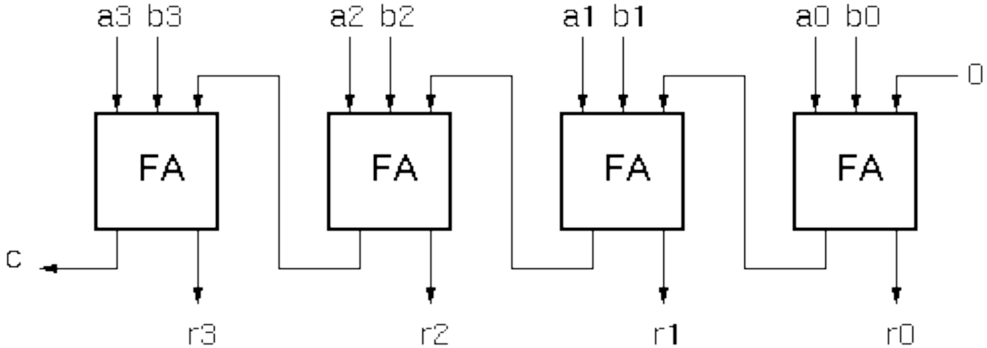
Review - Ripple Adder Example



```
module FullAdder(a, b, ci, r, co);
   input a, b, ci;
   output r, co;

   assign r = a ^ b ^ ci;
  assign co = a&ci + a&b + b&cin;

endmodule
```



```
module Adder(A, B, R);
  input [3:0] A;
  input [3:0] B;
  output [4:0] R;

  wire c1, c2, c3;
  FullAdder
  add0(.a(A[0]), .b(B[0]), .ci(1'b0), .co(c1),   .r(R[0]) ),
  add1(.a(A[1]), .b(B[1]), .ci(c1),    .co(c2),    .r(R[1]) ),
  add2(.a(A[2]), .b(B[2]), .ci(c2),    .co(c3),    .r(R[2]) ),
  add3(.a(A[3]), .b(B[3]), .ci(c3),    .co(R[4]),  .r(R[3]) );
endmodule
```

Example - Ripple Adder **Generator**

Parameters give us a way to generalize our designs. A module becomes a “generator” for different variations. Enables design/module reuse. Can simplify testing.

```
module Adder(A, B, R);
  parameter N = 4;
  input [N-1:0] A;
  input [N-1:0] B;
  output [N:0] R;
  wire [N:0] C;

  genvar i;

  generate
    for (i=0; i<N; i=i+1) begin:bit
      FullAdder add(.a(A[i], .b(B[i]), .ci(C[i]), .co(C[i+1]), .r(R[i]));
   end
  endgenerate

  assign C[0] = 1’b0;
  assign R[N] = C[N];
endmodule
```

Declare a parameter with default value.

Note: this is not a port. Acts like a “synthesis-time” constant.

Replace all occurrences of “4” with “N”.

variable exists only in the specification - not in the final circuit.

Keyword that denotes synthesis-time operations

For-loop creates instances (with unique names)

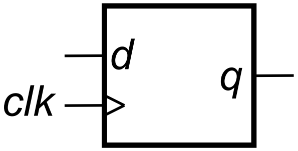
```
Adder adder4 ( ... );

Adder #(.N(64))
adder64 ( ... );
```

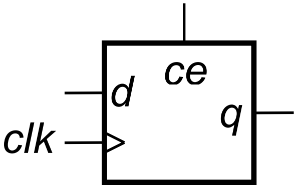
Overwrite parameter N at instantiation.

EECS151 Registers

- All registers are “N” bits wide - the value of N is specified at instantiation
- All positive edge triggered.

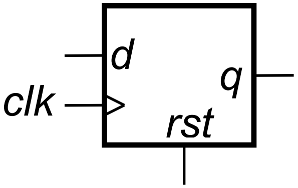


```
module REGISTER(q, d, clk);
    parameter N = 1;
```



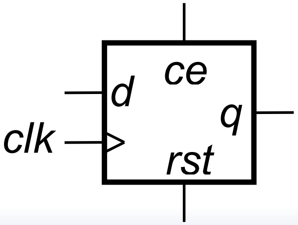
```
module REGISTER_CE(q, d, ce, clk);
    parameter N = 1;
```

On the rising clock edge if clock enable (ce) is 0 then the register is disabled (it’s state will not be changed).



```
module REGISTER_R(q, d, rst, clk);
    parameter N = 1;
    parameter INIT = 1b’0;
```

On the rising clock edge if reset (rst) is 1 then the state is set to the value of INIT. Default INIT value is all 0’s.

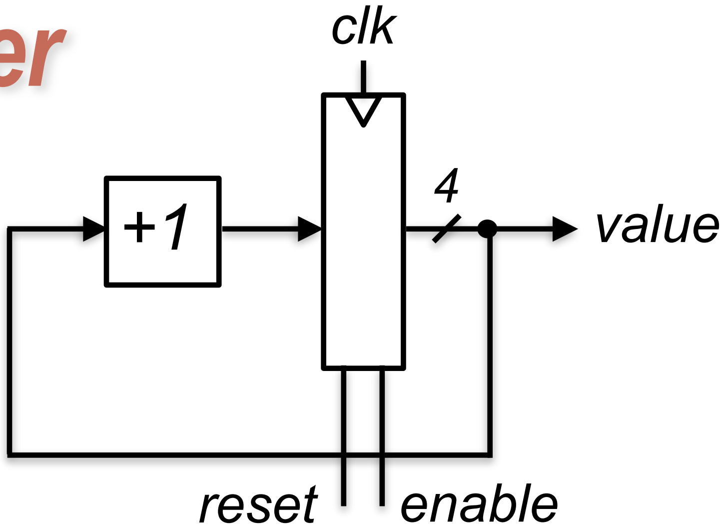


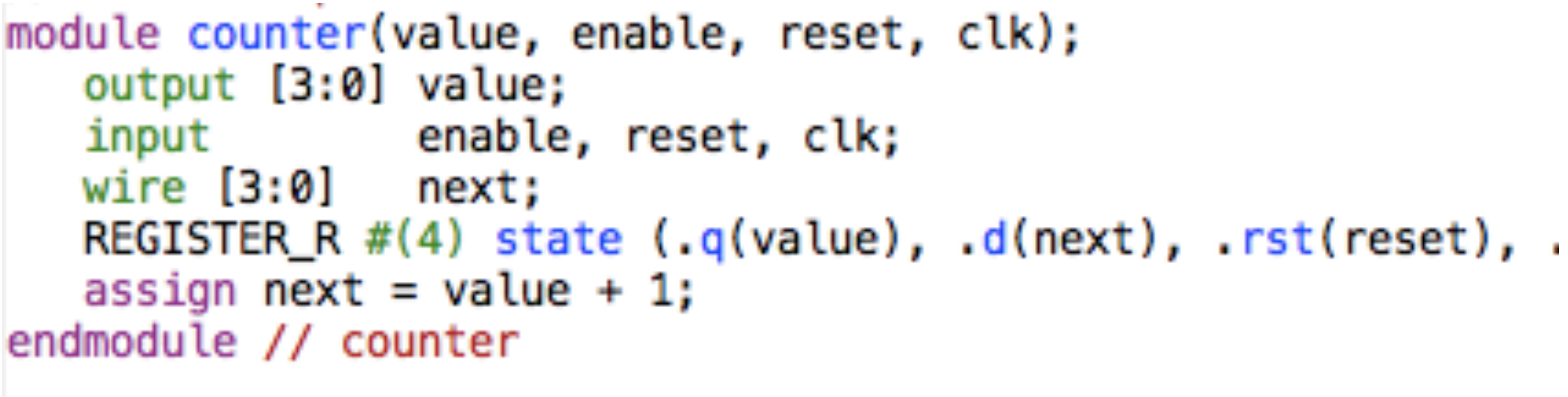
```
module REGISTER_R_CE(q, d, rst, ce, clk);
    parameter N = 1;
    parameter INIT = 1b’0;
```

Reset (rst) has priority over clock enable (ce).

4-bit wrap-around counter

0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 0, 1, …

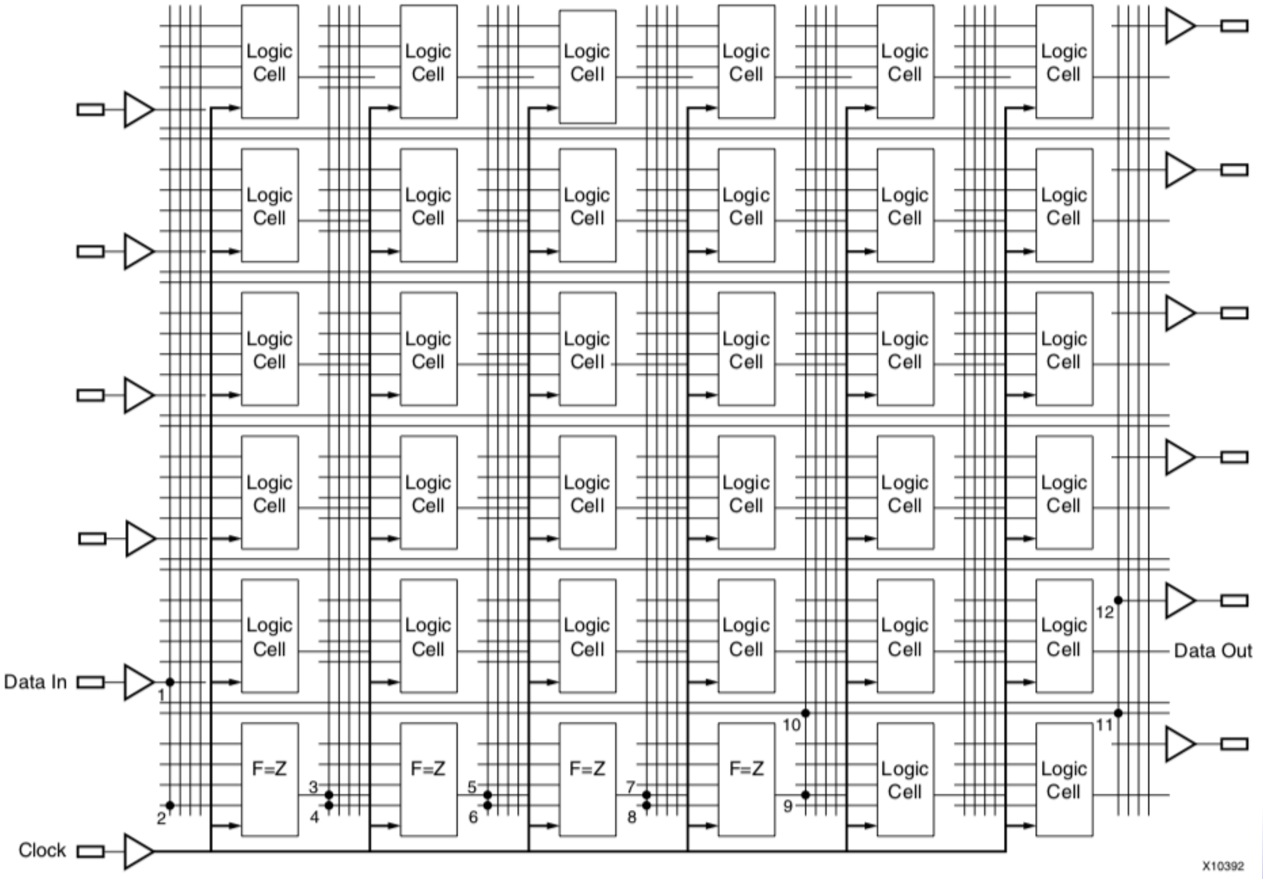




```
module counter(value, enable, reset, clk);
   output [3:0] value;
   input        enable, reset, clk;
   wire [3:0]   next;
   REGISTER_R #(4) state (.q(value), .d(next), .rst(reset), .
   assign next = value + 1;
endmodule // counter
```

FPGA Overview

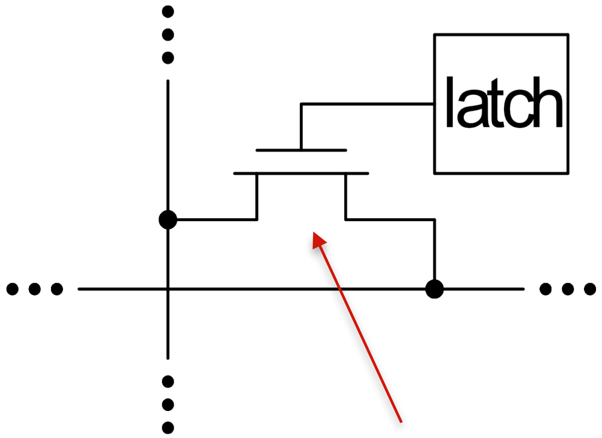
- Basic idea: two-dimensional array of logic blocks and flip-flops with a means for the user to configure (program):
  1. the interconnection between the logic blocks,
  2. the function of each block.



Simplified version of FPGA internal architecture

User Programmability

- Latch-based (Xilinx, Intel/Altera, …)



MOSFET used as a “switch”

\+ reconfigurable

– volatile

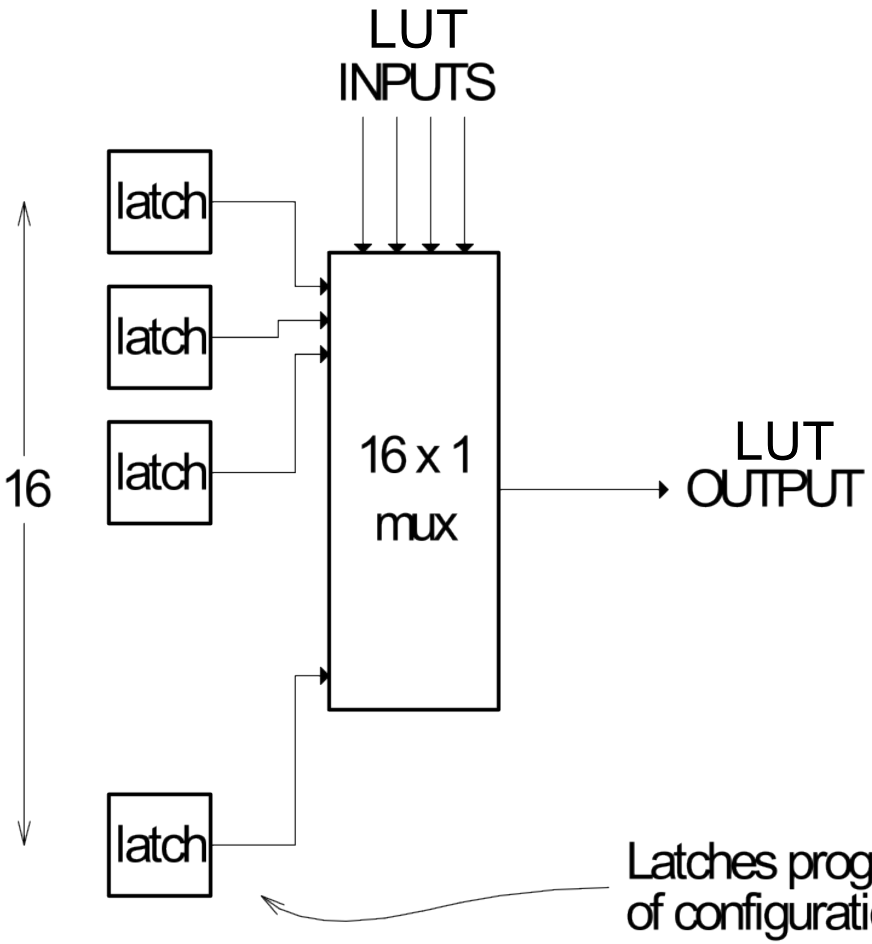
– relatively large.

❑ 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

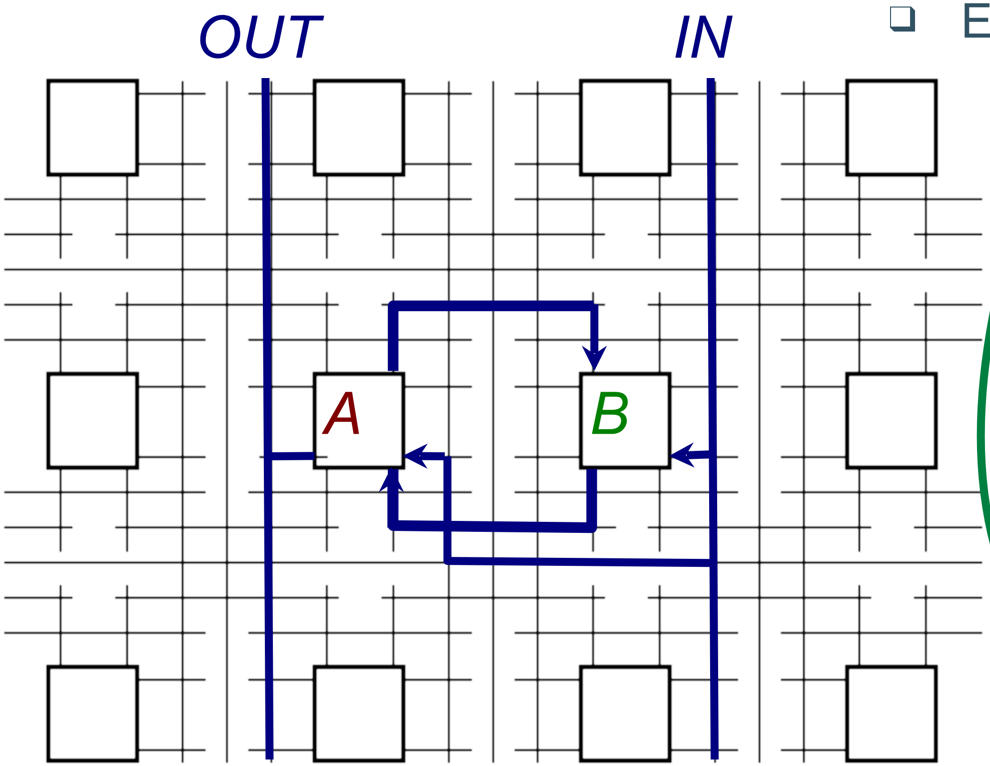
4-LUT Implementation



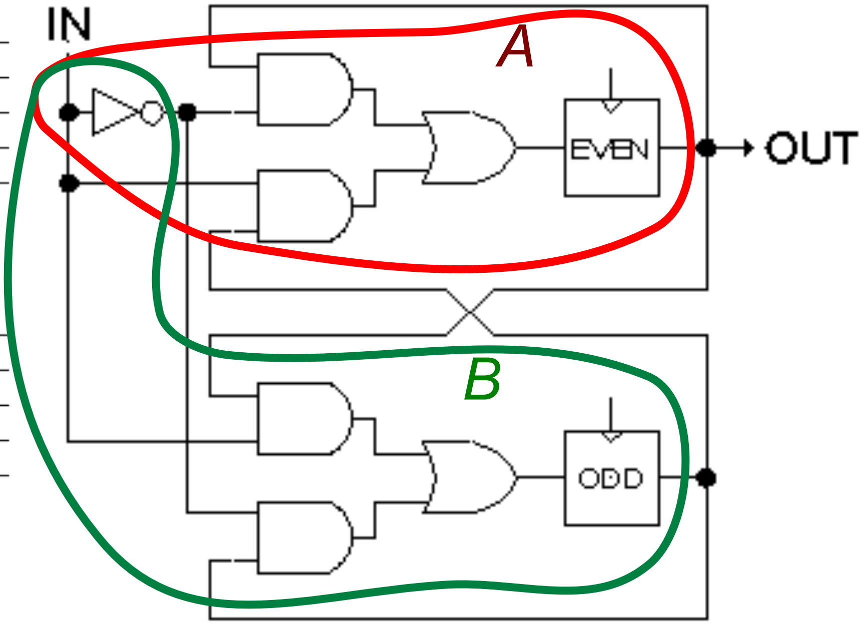
Latches programmed as part of configuration bit-stream

- 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!

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.

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

Involution Law:

(x')' = x

Laws of Complementarity:

x + x' = 1   x x' = 0

Commutative Law:

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

Algebraic Simplification

$$\begin{aligned}
Cout &= a'bc + ab'c + abc' + abc \\
&= a'bc + ab'c + abc' + abc + abc \\
&= a'bc + abc + ab'c + abc' + abc \\
&= (a' + a)bc + ab'c + abc' + abc \\
&= (1)bc + ab'c + abc' + abc \\
&= bc + ab'c + abc' + abc + abc \\
&= bc + ab'c + abc + abc' + abc \\
&= bc + a(b' + b)c + abc' + abc \\
&= bc + a(1)c + abc' + abc \\
&= bc + ac + ab(c' + c) \\
&= bc + ac + ab(1) \\
&= bc + ac + ab
\end{aligned}$$

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)**
- <u>Sum of Products</u> (disjunctive normal form, <u>minterm</u> 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:

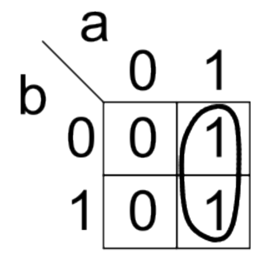
`f = a'bc + ab'c' + ab'c + abc' + abc`

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

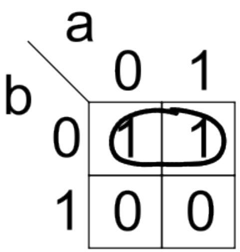
What is the cost?

Karnaugh Map Method

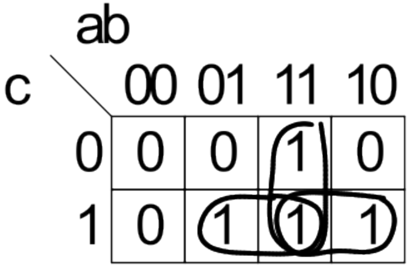
- Adjacent groups of 1's represent product terms



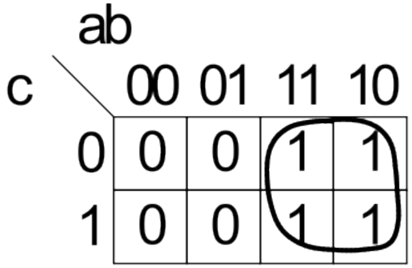
$f = a$



$g = b'$



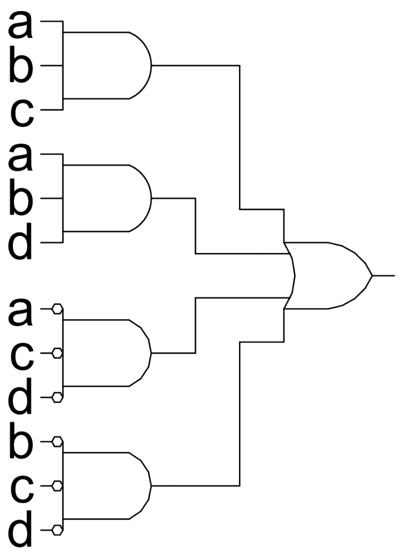
$cout = ab + bc + ac$



$f = a$

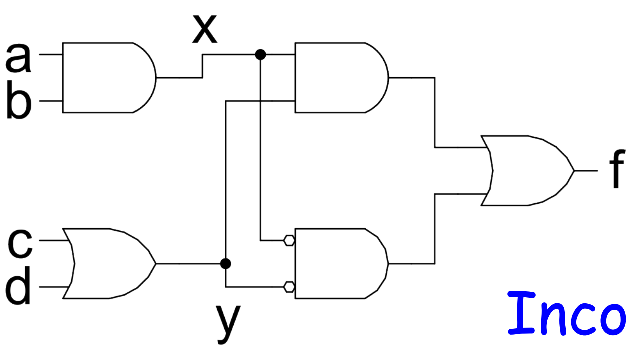
Multi-level Combinational Logic

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:

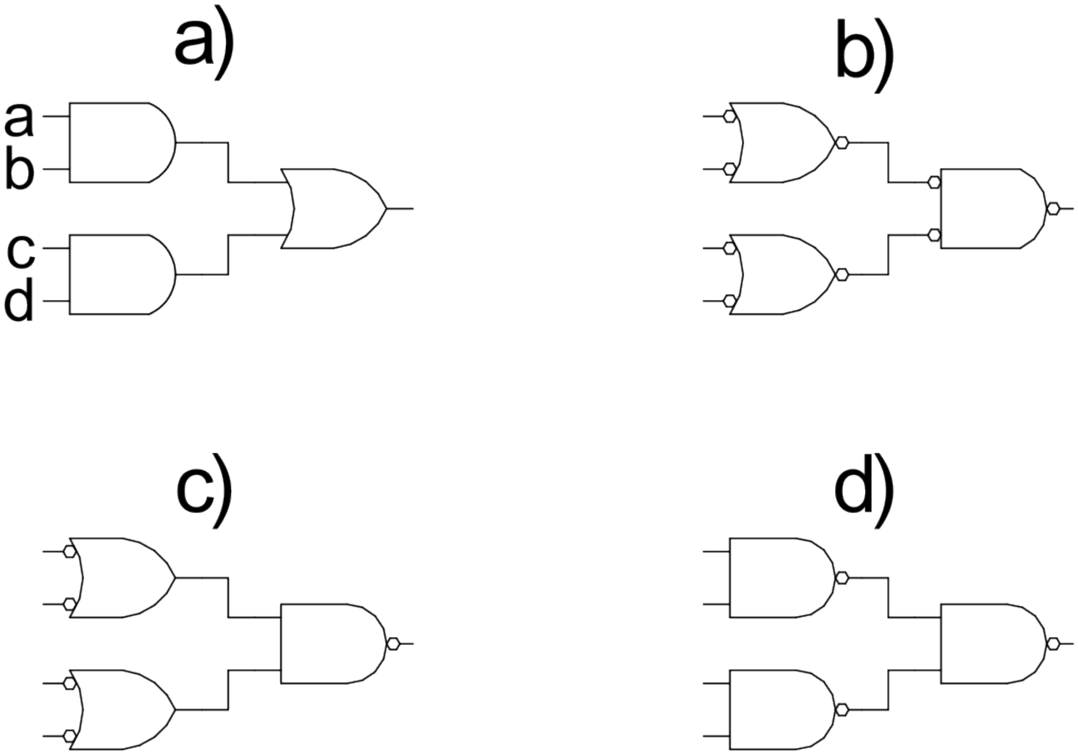
a) CAD Tools use sophisticated algorithms and heuristics

Guess what? These problems tend to be NP-complete

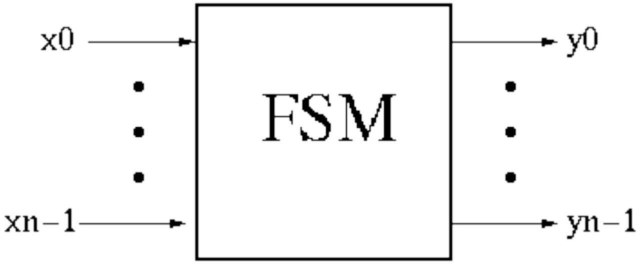
b) Humans and tools often exploit some special structure (example adder)

NAND-NAND & NOR-NOR Networks

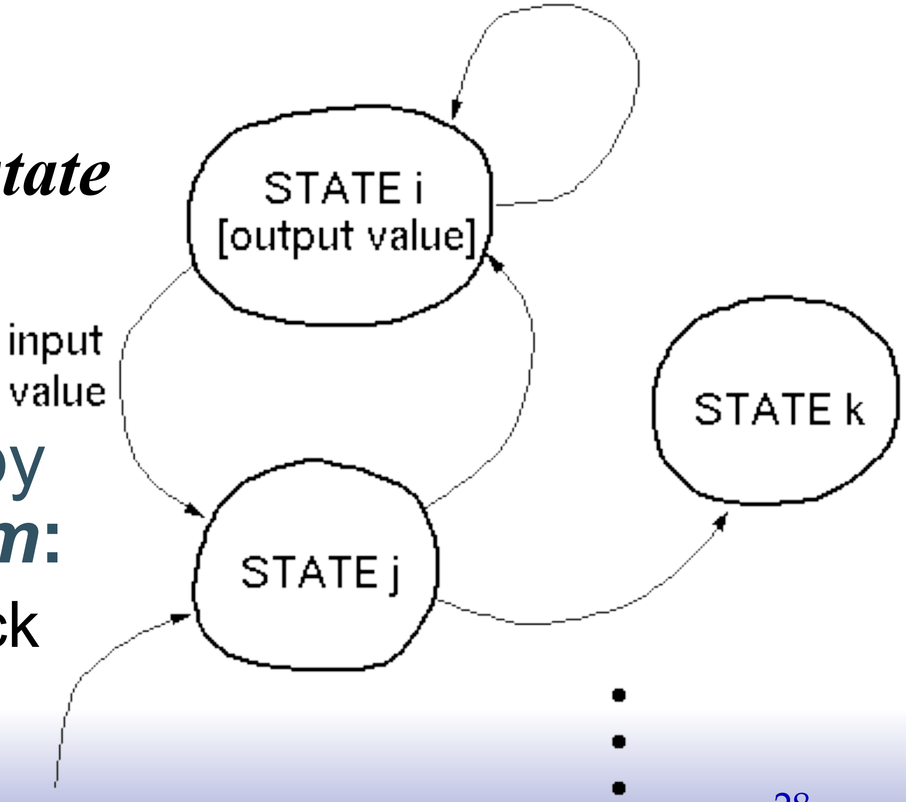
- Mapping from AND/OR to NAND/NAND



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***

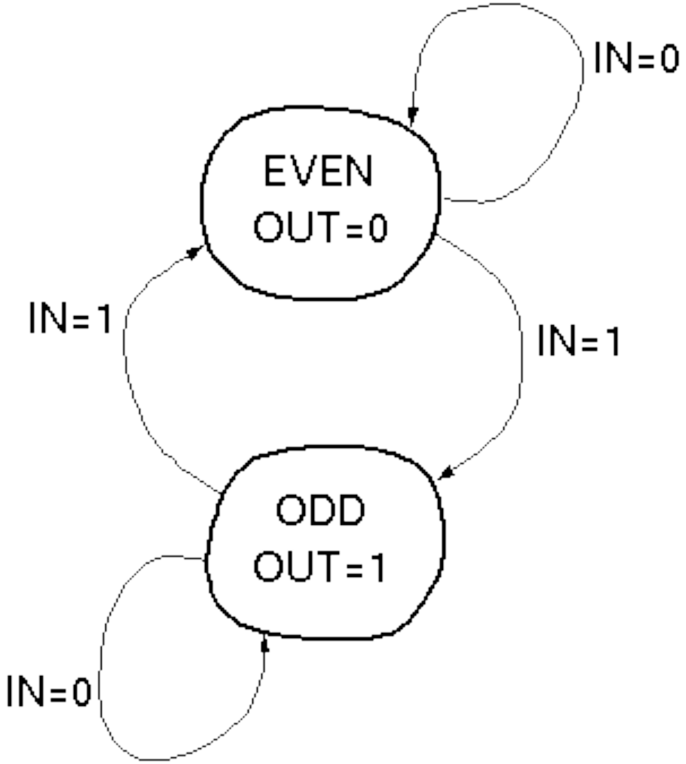


- Behavior is represented by ***State Transition Diagram***:
  - traverse one edge per clock cycle.

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

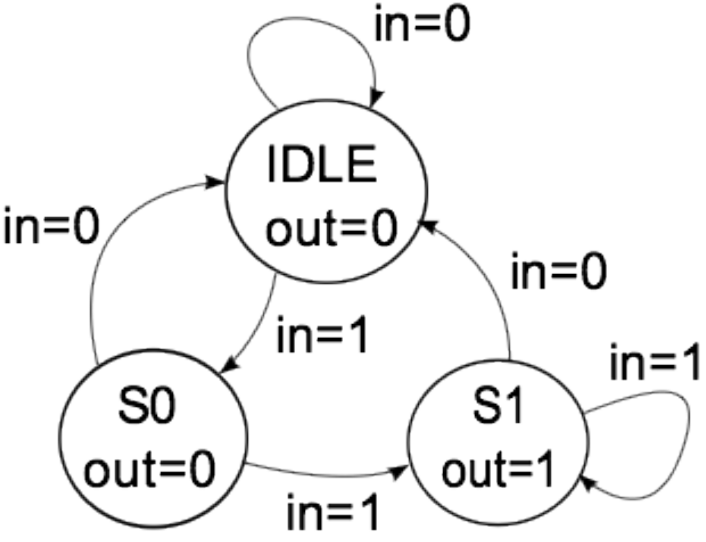
present state (ps)	*OUT*	*IN*	*next state (ns)*
0	*0*	*0*	*0*
0	*0*	*1*	*1*
1	*1*	*0*	*1*
1	*1*	*1*	*0*

Derive logic equations from table (how?):

OUT = PS

NS = PS xor IN

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***

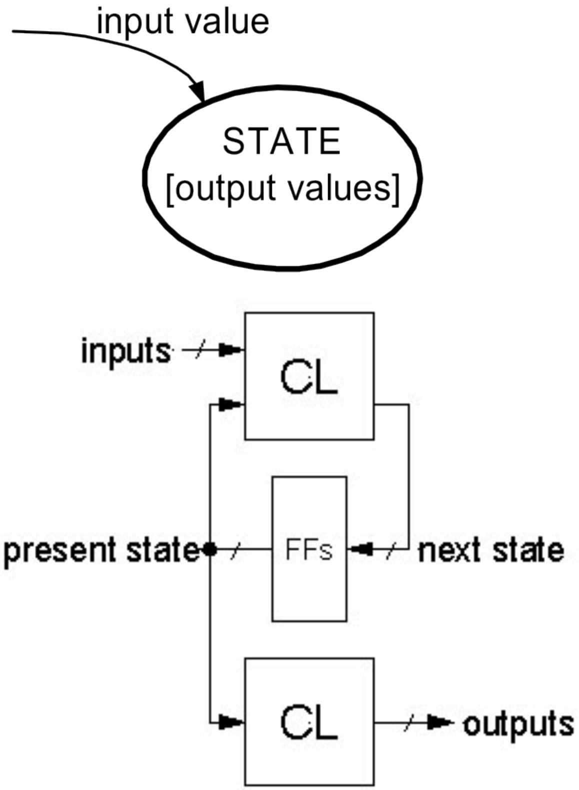
Normal values: used unless specified below.

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

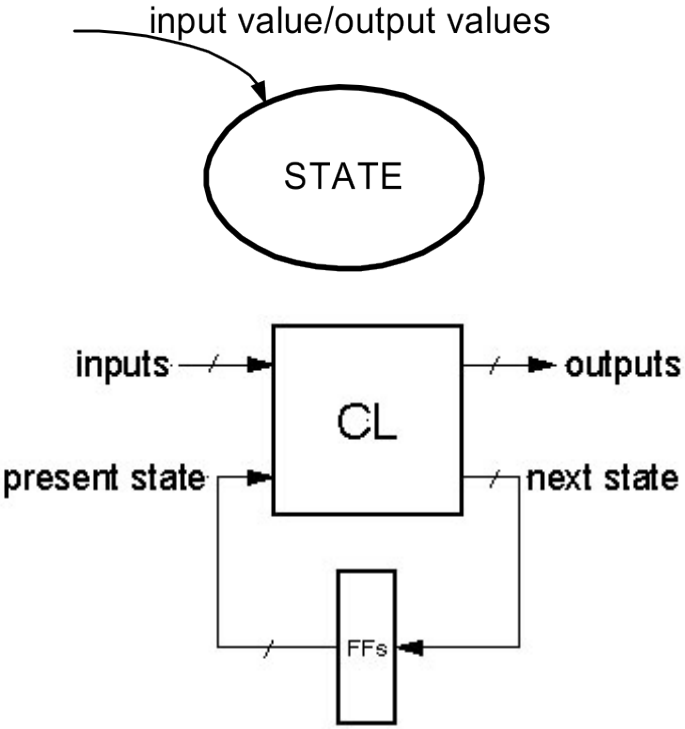
Note: The use of "blocking assignments" allow signal values to be "rewritten", simplifying the specification.

FSM Recap

Moore Machine

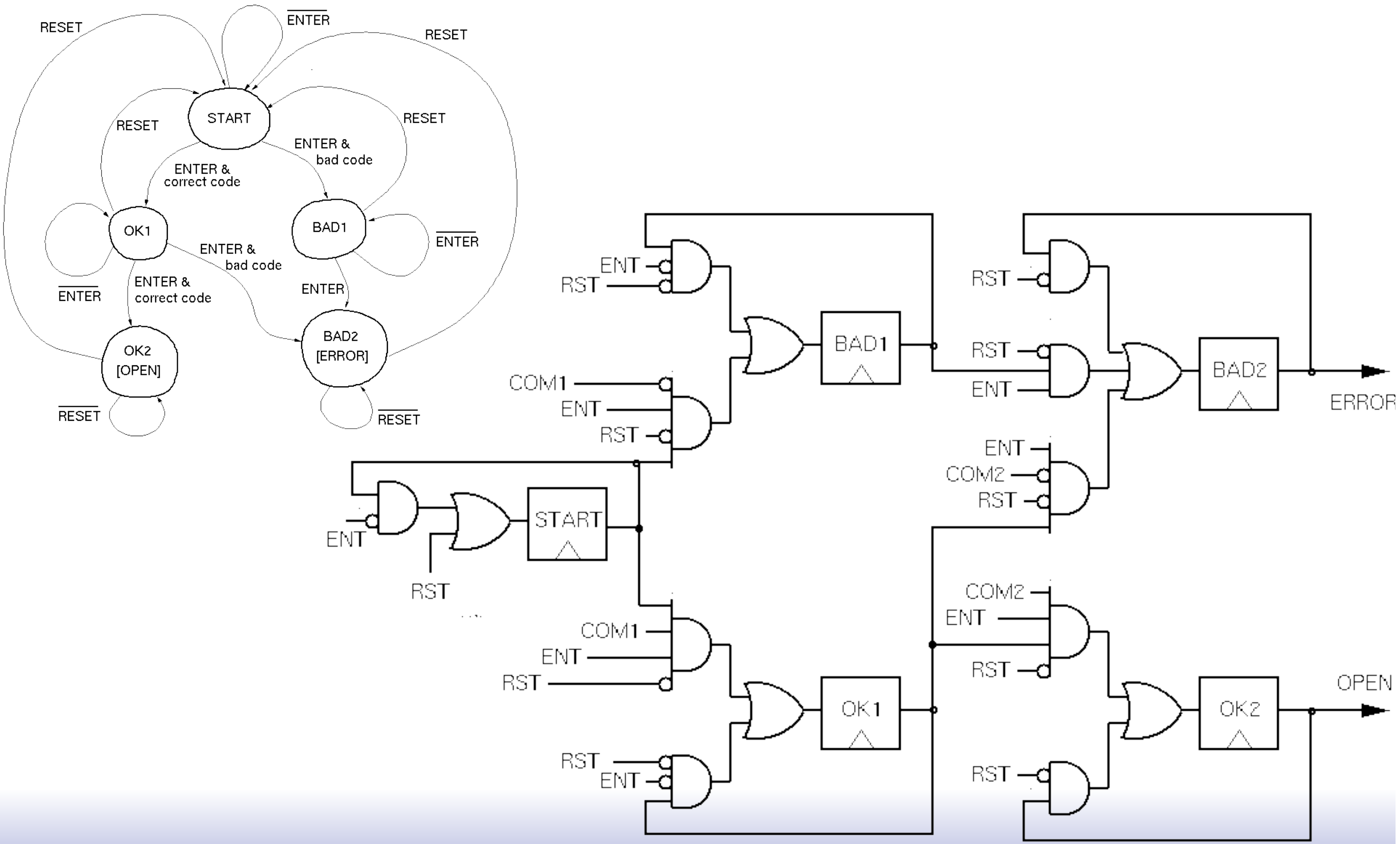


Mealy Machine

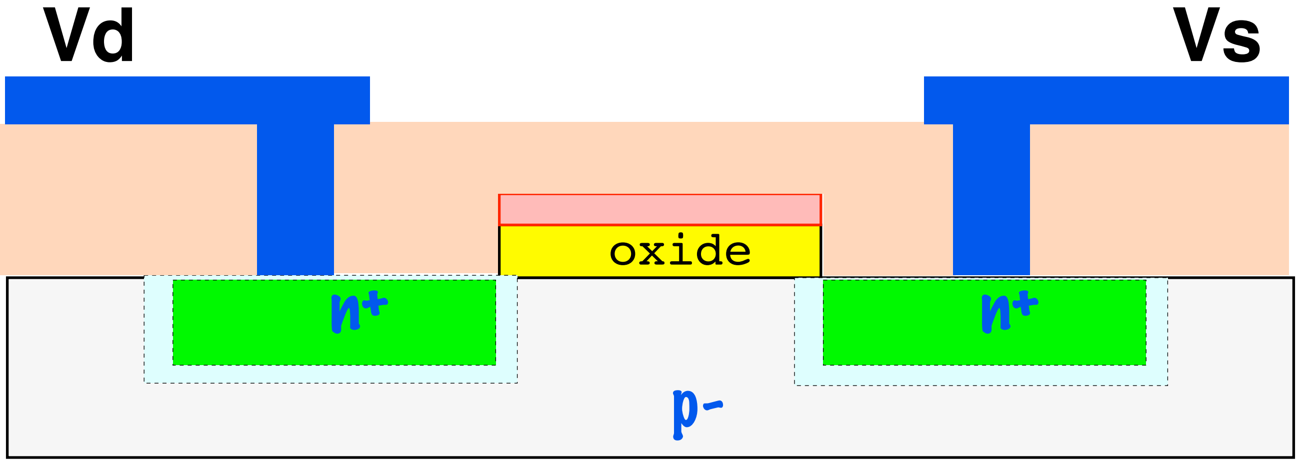


Both machine types allow one-hot implementations.

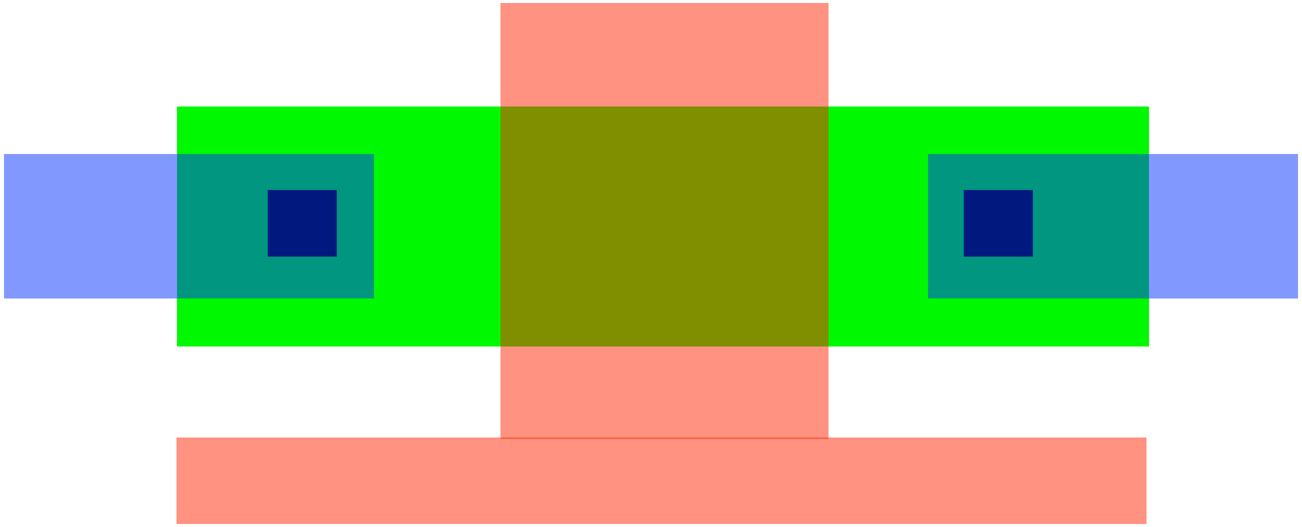
One-hot encoded combination lock



Final product ...



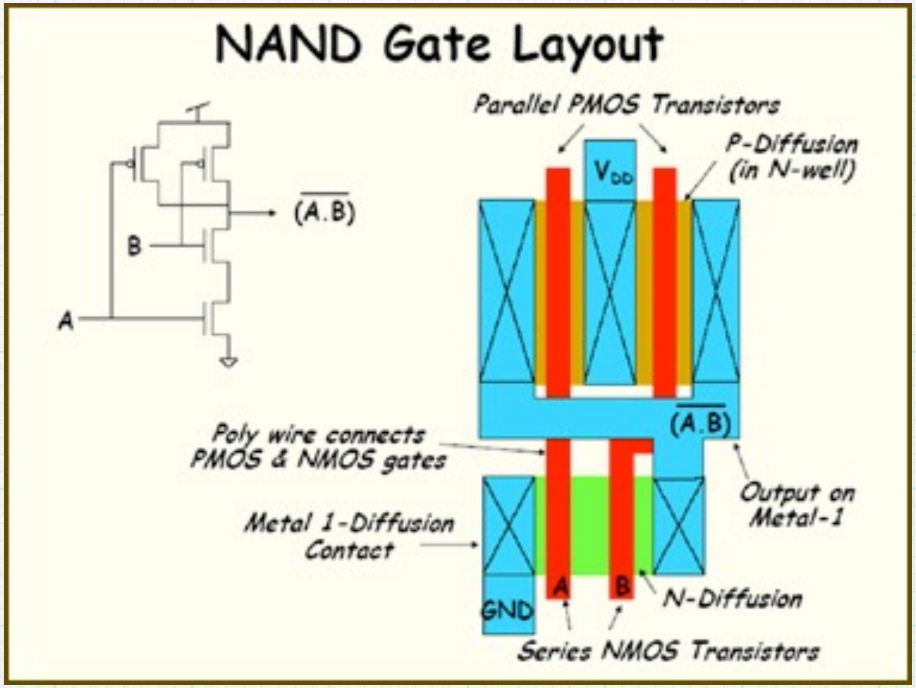
Top-down view:



“The planar process”

Jean Hoerni,
Fairchild
Semiconductor
1958

Physical Layout

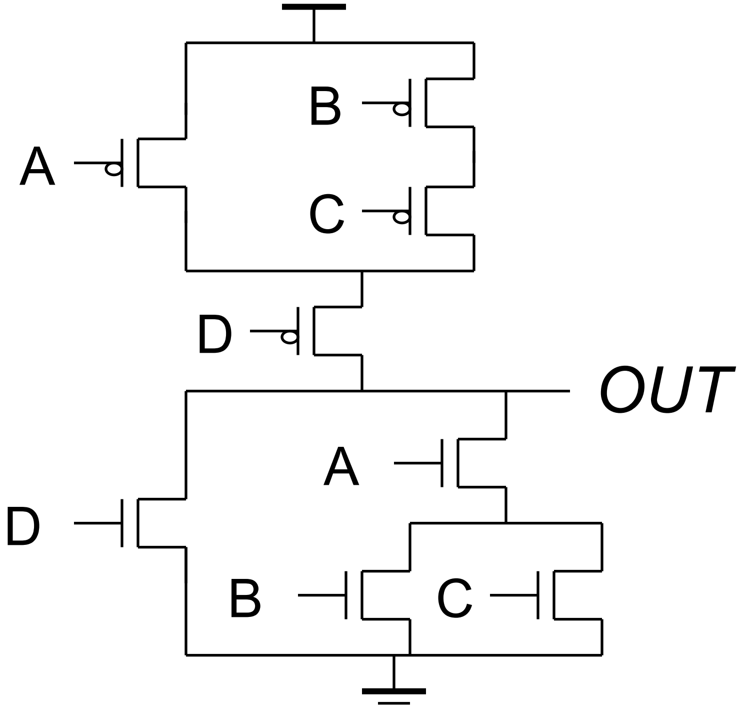


- How do transistor circuits get "laid out" as geometry?
- What circuit does a physical layout implement?
- Where are the transistors and wires and contacts and vias?

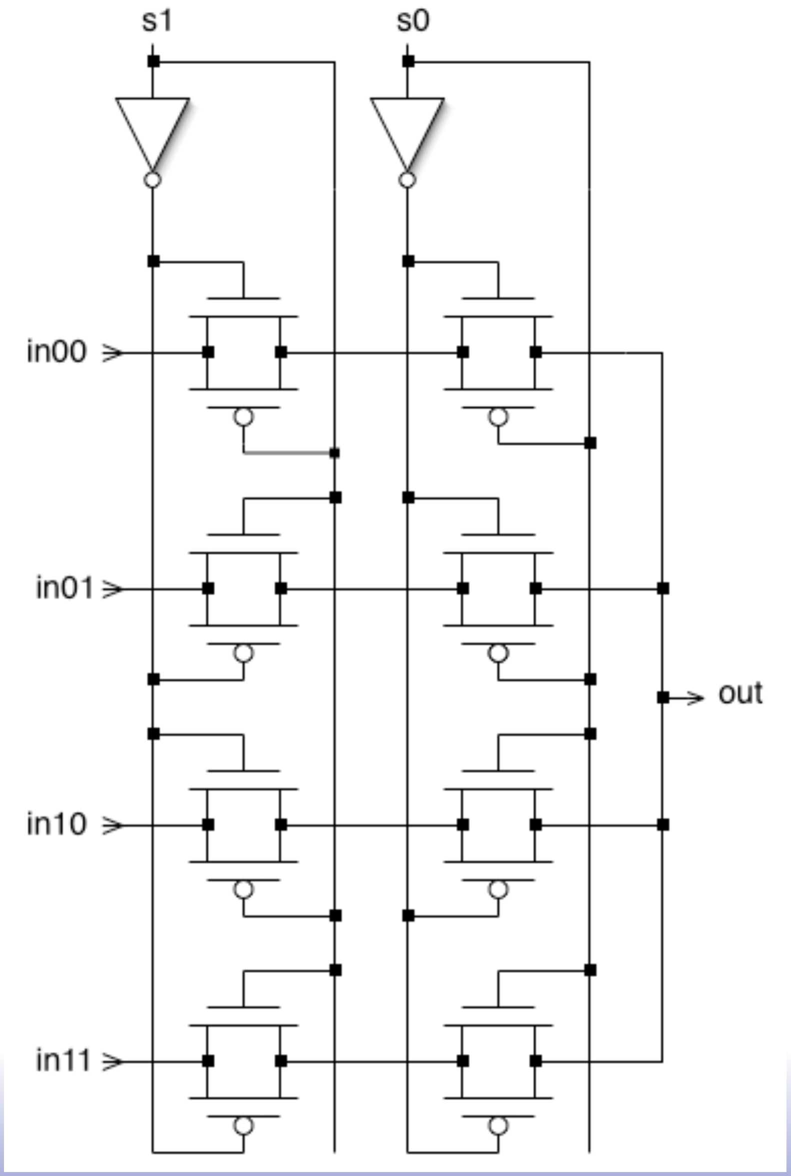
Complex CMOS Gate

$\text{OUT} = \overline{D + A \cdot (B + C)}$

$\text{OUT} = \overline{D \cdot A + B \cdot C}$



4-to-1 Transmission-gate Mux

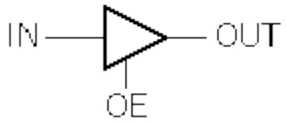


- The series connection of pass-transistors in each branch effectively forms the AND of s1 and s0 (or their complement).
- Compare cost to logic gate implementation

Any better solutions?

Tri-state Buffers

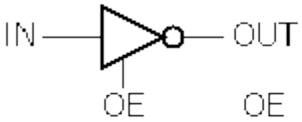
Tri-state Buffer:



OE	IN	OUT
0	0	Z
0	1	Z
1	0	0
1	1	1

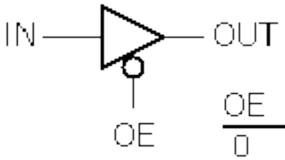
"high impedance" (output disconnected)

Variations:



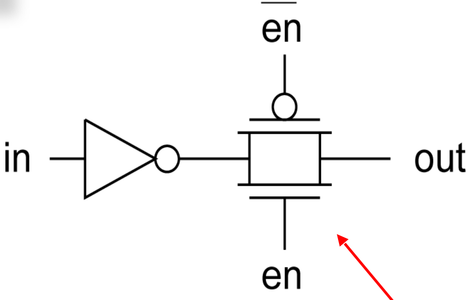
OE	IN	OUT
0	–	Z
1	0	1
1	1	0

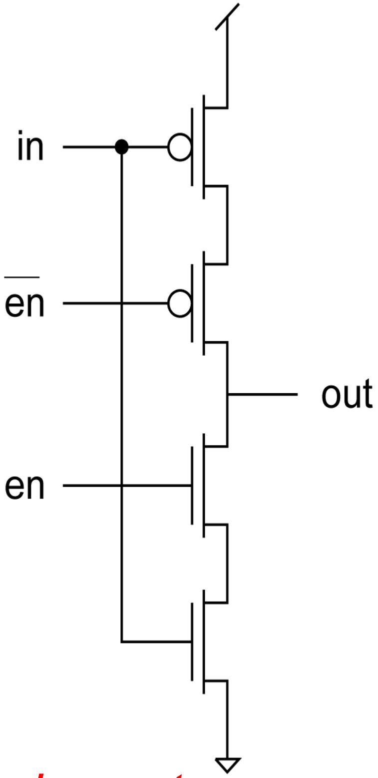
Inverting buffer



OE	IN	OUT
0	0	0
0	1	1
1	–	Z

Inverted enable

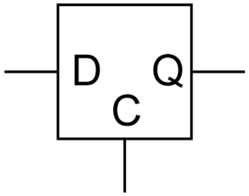


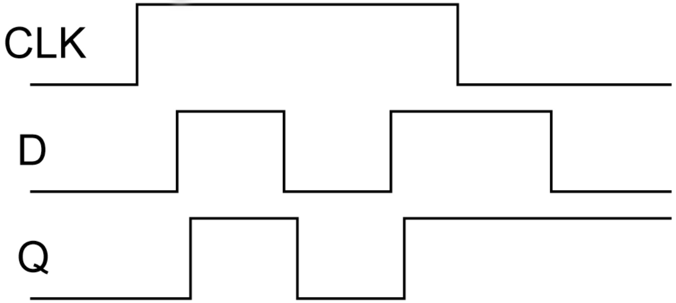


transmission gate useful in implementation

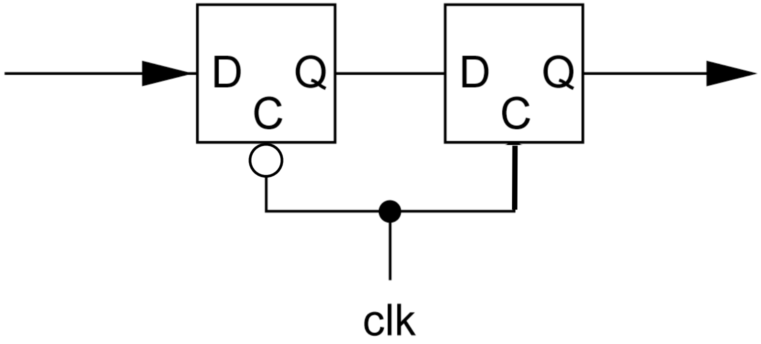
Latches and Flip-flops

Positive Level-sensitive ***latch***:

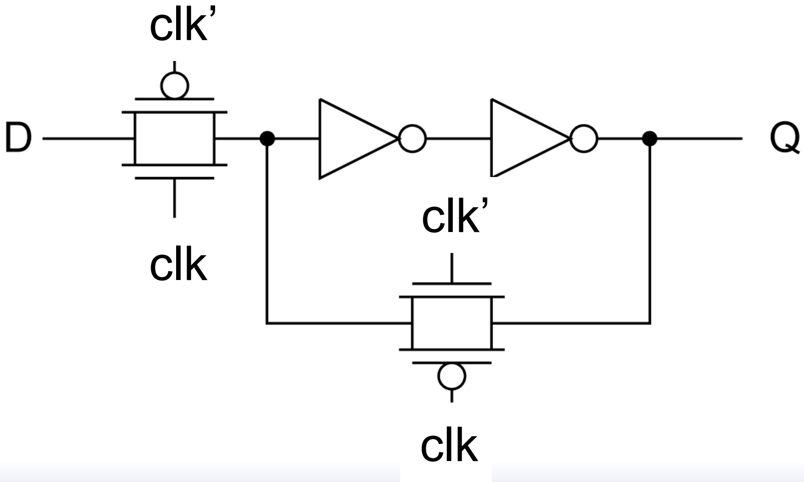




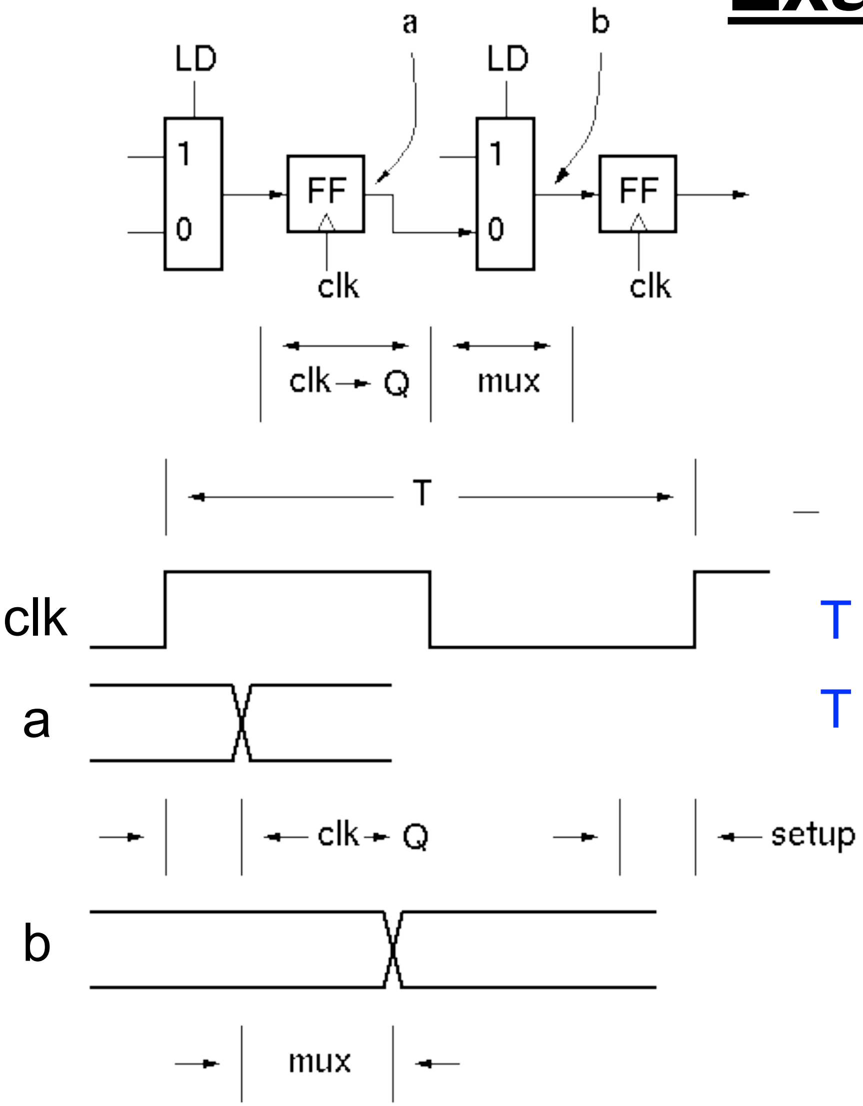
Positive Edge-triggered **flip-flop** built from two level-sensitive latches:



Latch Implementation:



Example

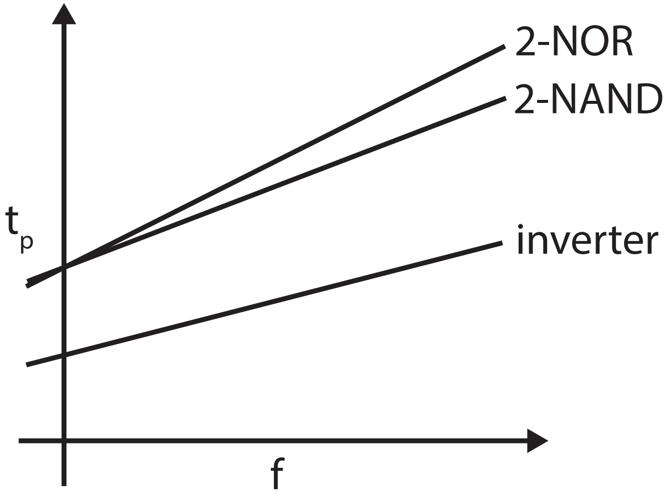


Parallel to serial converter circuit

$T \geq \text{time}(\text{clk}\rightarrow\text{Q}) + \text{time}(\text{mux}) + \text{time}(\text{setup})$

$T \geq \tau_{\text{clk}\rightarrow\text{Q}} + \tau_{\text{mux}} + \tau_{\text{setup}}$

Gate Delay Summary



The y-intercepts (intrinsic delay) for NAND and NOR are both twice that of the inverter. The NAND line has a gradient 4/3 that of the inverter (steeper); for NOR it is 5/3 (steepest).

$$t_{p0}\left(2+\frac{4f}{3\gamma}\right)$$

2-input NAND

$$t_{p0}\left(2+\frac{5f}{3\gamma}\right)$$

2-input NOR

What about gates with more than 2-inputs?

4-input NAND:

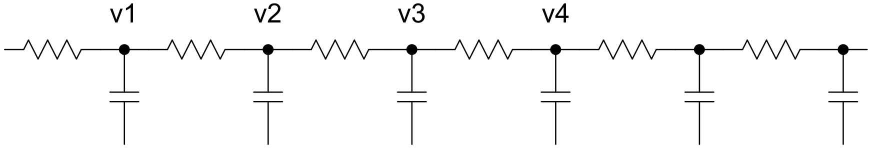
$$t_p = t_{p0}\left(4+\frac{2f}{\gamma}\right)$$

slope

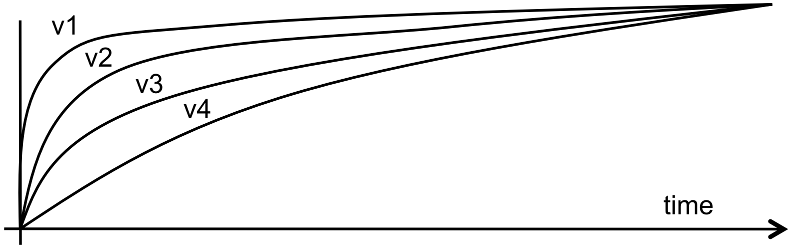
intercept

Wire Delay

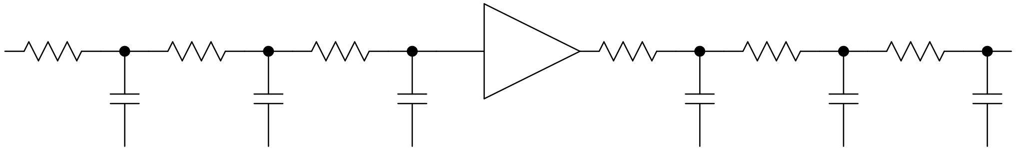
- Even in those cases where the transmission line effect is negligible:
  - Wires posses distributed resistance and capacitance



  - Time constant associated with distributed RC is proportional to the *square* of the length



- For **short wires** on ICs, resistance is insignificant (relative to effective R of transistors), but C is important.
  - Typically around half of C of gate load is in the wires.
- For **long wires** on ICs:
  - busses, clock lines, global control signal, etc.
  - Resistance is significant, therefore distributed RC effect dominates.
  - signals are typically "rebuffered" to reduce delay:



How to retime logic

Circles are combinational logic, labelled with delays.

Critical path is 5. We want to improve it without changing circuit semantics.

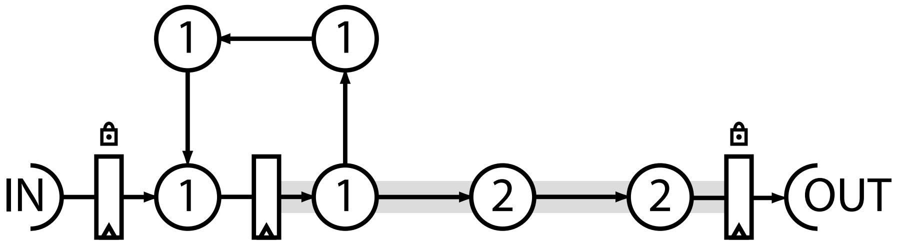


Figure 1: A small graph before retiming. The nodes represent logic delays, with the inputs and outputs passing through mandatory, fixed registers. The critical path is 5.

Add a register, move one circle. Performance improves by 20%.

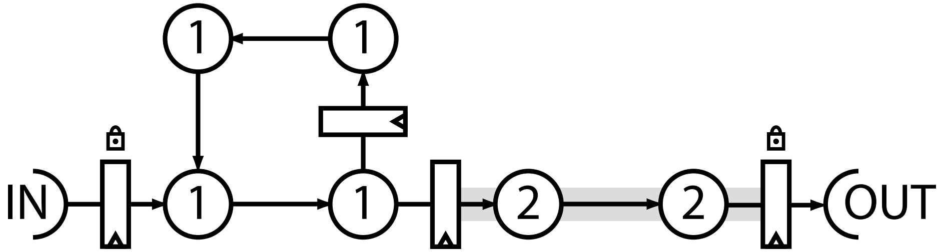
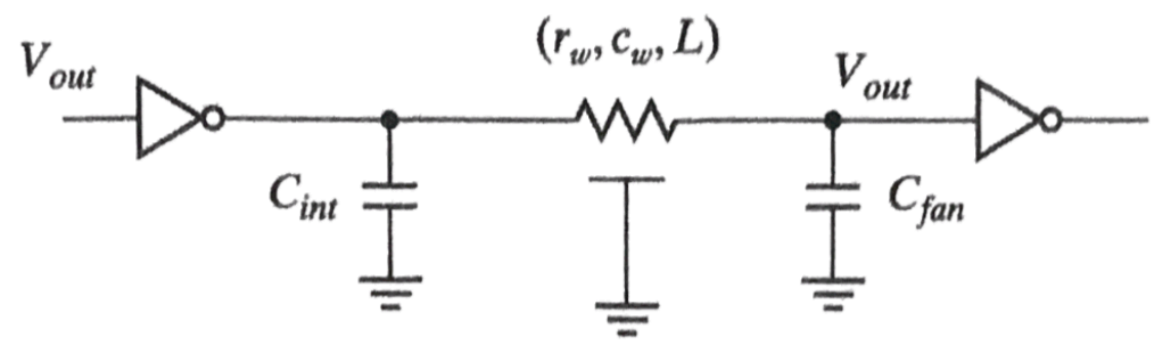


Figure 2: The example in Figure 2 after retiming. The critical path is reduced from 5 to 4.

Logic Synthesis tools can do this in simple cases.

Gate Driving long wire and other gates



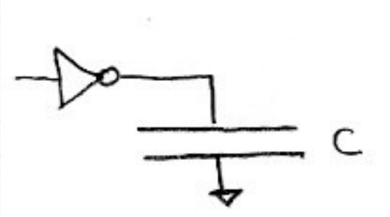
$$R_w = r_w L, \quad C_w = c_w L$$

$$t_p = 0.69 R_{dr} C_{int} + 0.69 R_{dr} C_w + 0.38 R_w C_w + 0.69 R_{dr} C_{fan} + 0.69 R_w C_{fan}$$

$$= 0.69 R_{dr}(C_{int} + C_{fan}) + 0.69(R_{dr} c_w + r_w C_{fan}) L + 0.38 r_w c_w L^2$$

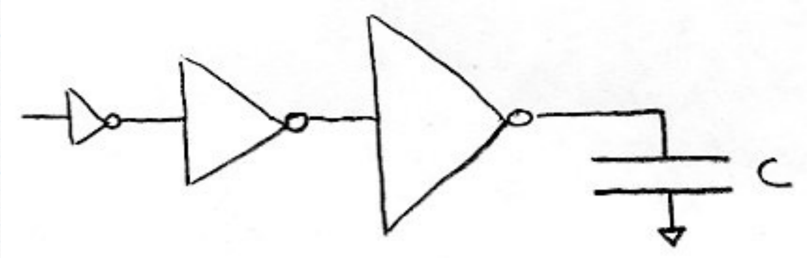
Driving Large Loads

- Large fanout nets: clocks, resets, memory bit lines, off-chip
- Relatively small driver results in long rise time (and thus large gate delay)



- Strategy:

Staged Buffers



- How to optimally scale drivers?
- Optimal trade-off between delay per stage and total number of stages?