EECS 151/251A
Spring 2019
Digital Design and Integrated Circuits

Instructor:
Wawrzynek

Lecture 14

Midterm Exam Review

The exam will take place Thursday March 14, 6–9PM in 306 Soda Hall. The exam comprises a set of questions with 1 point per expected minute of completion with a total of approximately 90 points. 251A students will be asked to complete extra questions. All students are allowed one 2-sided 8.5 × 11 inch sheet of notes. No calculators, phones, or other electronic devices will be allowed. Slide-rules will be permitted.

1. Moore's Law Definition and Consequences
2. Dennard Scaling and Consequences
3. Cost/Performance/Power Design Tradeoffs and Pareto Optimality
4. Definitions and representations of combinational logic
5. Principle of restoration
6. Basic principle behind edge-triggered clocking and RTL design methodology
7. Digital system implementation technology alternatives and relative strengths and weaknesses
8. FPGA versus ASIC cost analysis
9. Principle behind structural versus behavioral hardware description
10. Basic Verilog descriptions for combinational logic
11. Verilog generators blocks
12. Verilog inference of state elements
13. FPGA reconfigurable fabric architecture
14. Details of FPGA fabric interconnect switches
15. Details of FPGA logic blocks with LUT

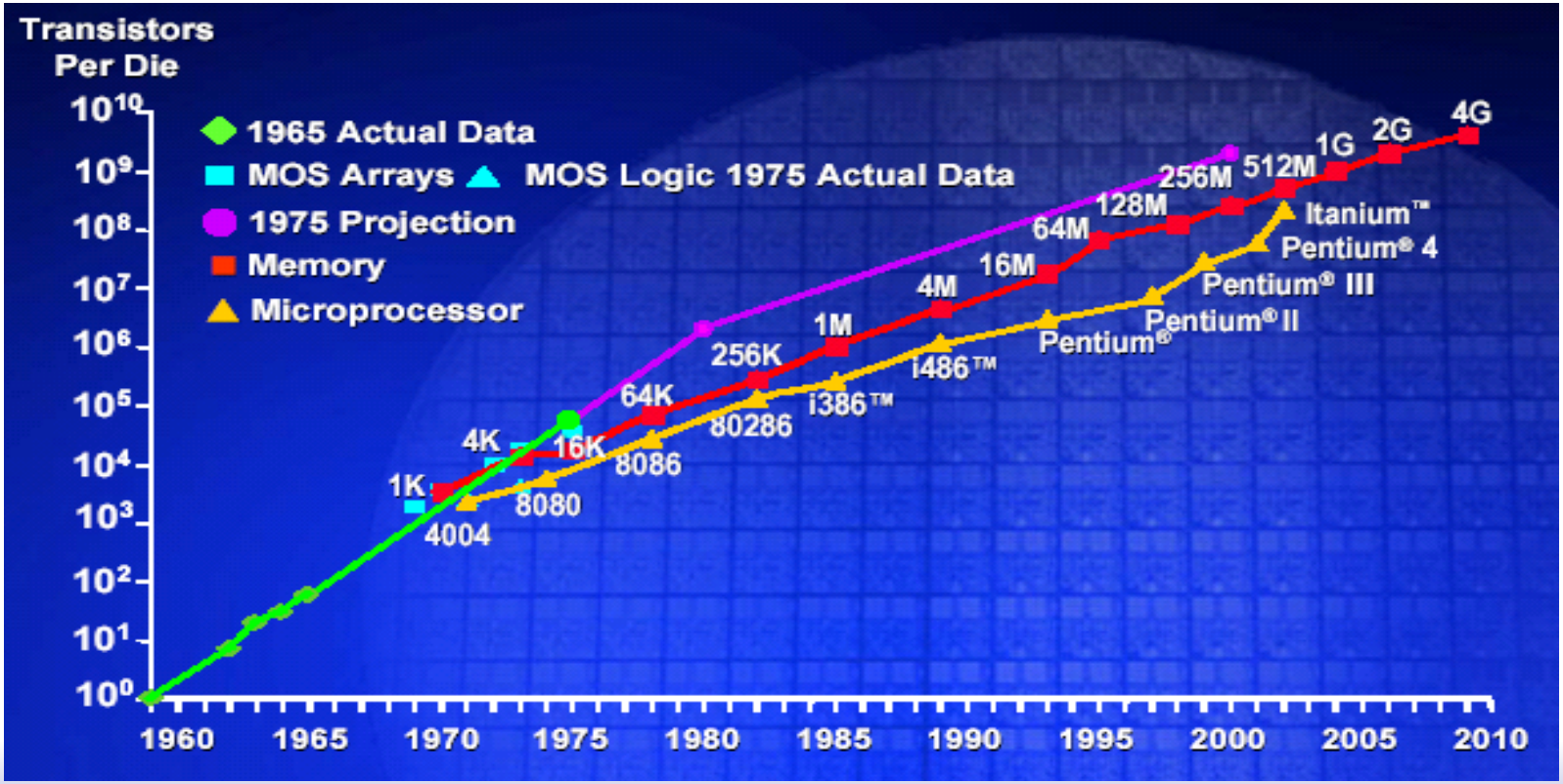
16. Logic circuit partitioning for and mapping to FPGA fabric
17. Laws of Boolean algebra
18. Boolean algebra representation of logic circuit and manipulation
19. Canonical forms
20. K-map method for 2-level logic simplification
21. Multi-level logic circuits
22. Bubble-pushing method for converting from AND/OR to NAND/NOR

23. FSM state transition diagram (STD) representation
24. FSM implementation in circuit based on STD
25. FSM one-hot encoding design method
26. FSMs description in Verilog
27. FSM Moore versus Mealy styles
28. Basics of planar CMOS IC processing
29. IC manufacturing trends and scaling
30. Static switch-level complementary CMOS logic gates
31. Transmission-gate logic circuits
32. Tri-state buffers
33. Edge-triggered Flip-flop implementation and operation
34. Maximum clock frequency calculation from circuit parameters
35. Origin of gate-delay and calculation
36. Delay property of wires and rebuffering
37. Circuit register rebalancing
38. Logic delay combined with wires
39. Driving large capacitive loads

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.

What we get:
κ^2 as many transistors at the same power density!

Whose gates switch κ times faster!

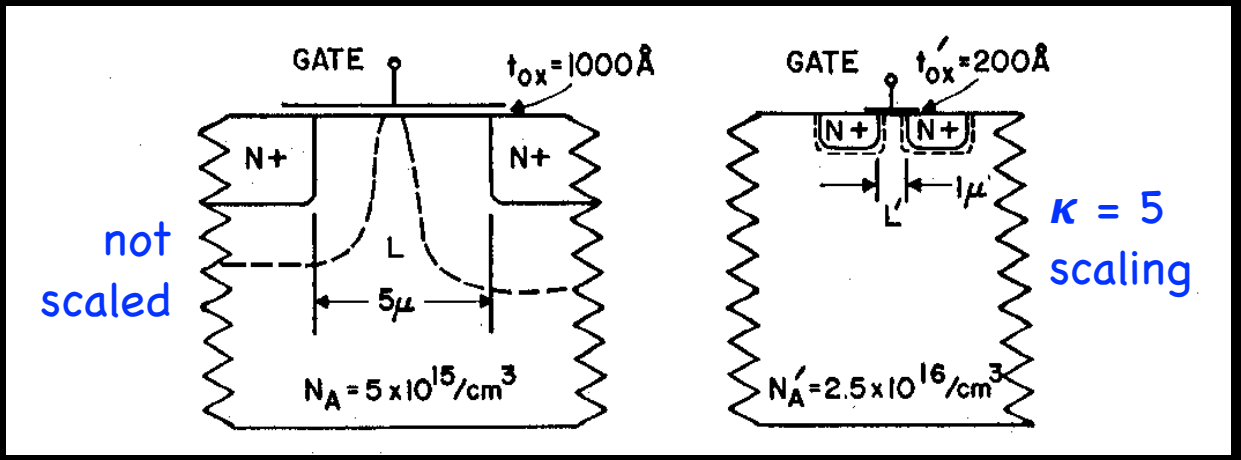


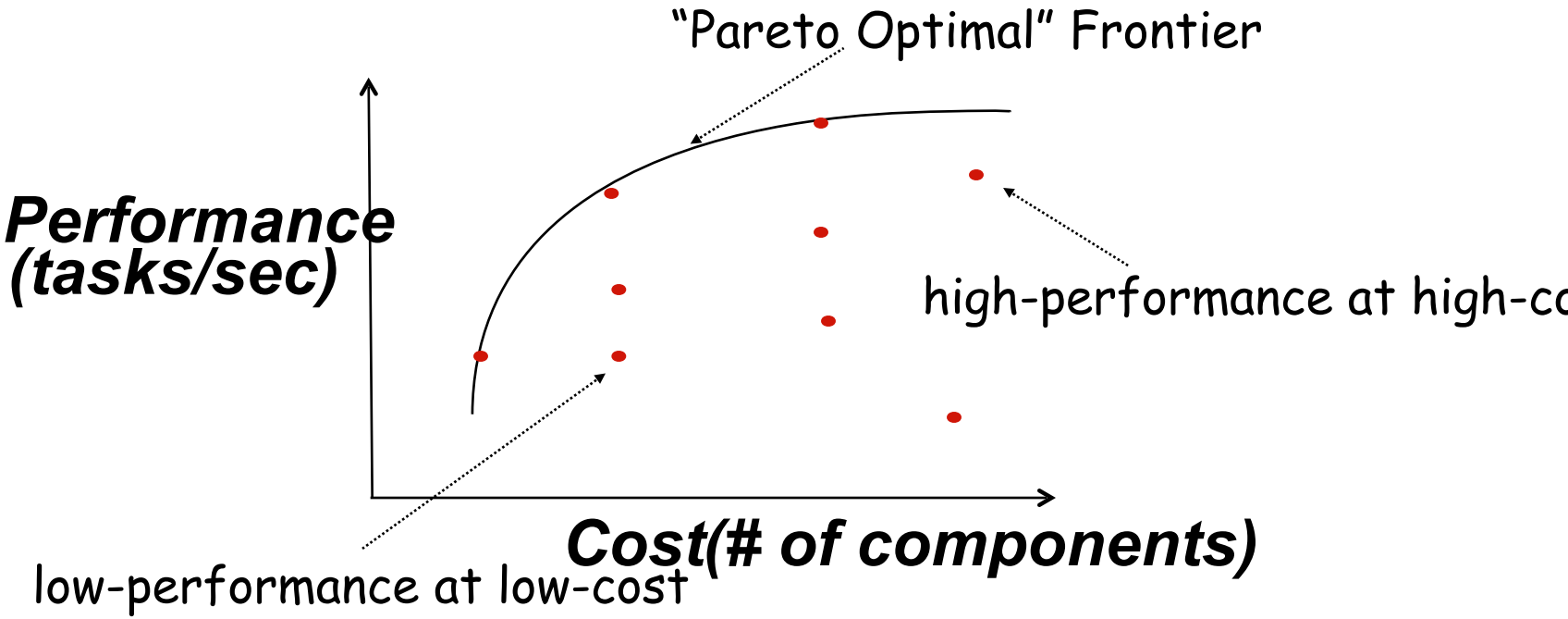
TABLE I

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

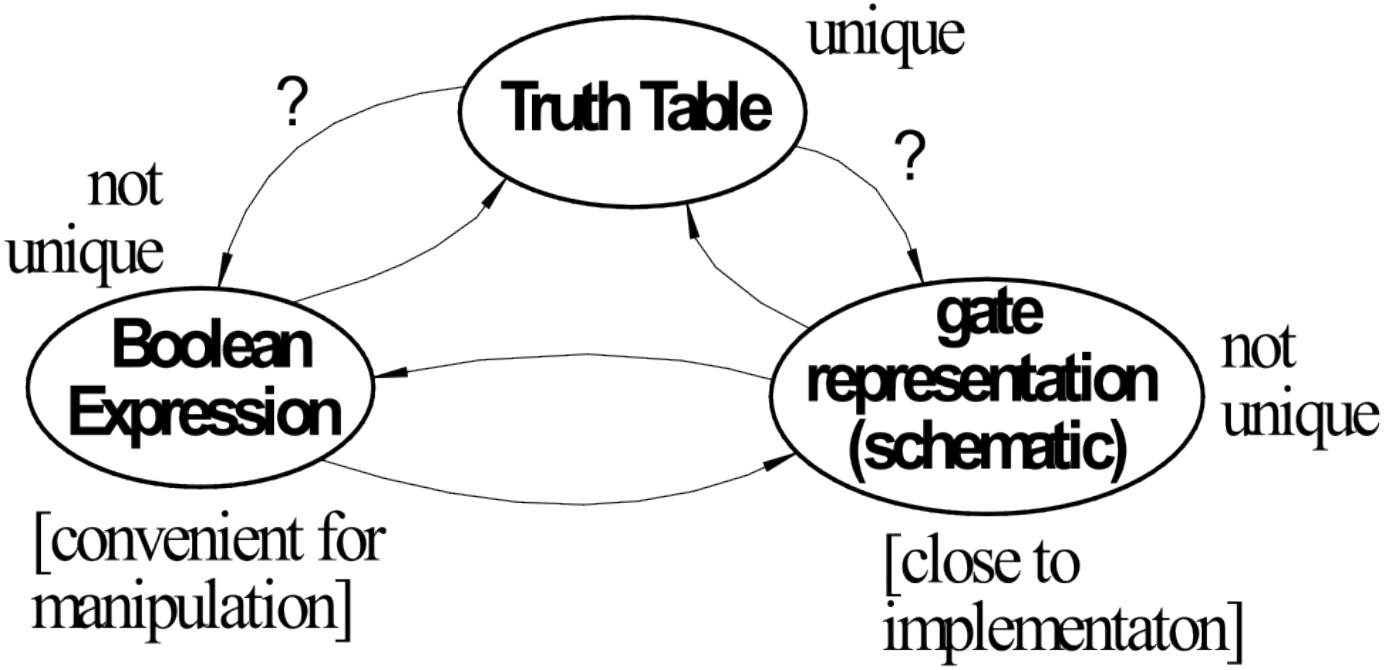
Power density scaling ended in 2003 (Pentium 4: 3.2GHz, 82W, 55M FETs).

Design Space & Optimality



Relationship Among Representations

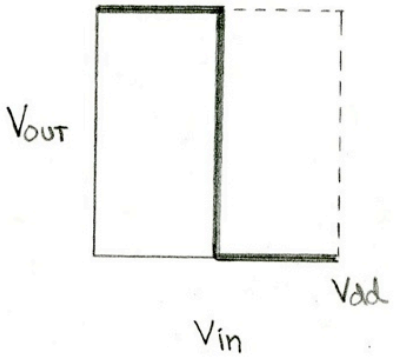
* Theorem: Any Boolean function that can be expressed as a truth table can be written as an expression in Boolean Algebra using AND, OR, NOT.



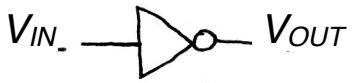
How do we convert from one to the other?

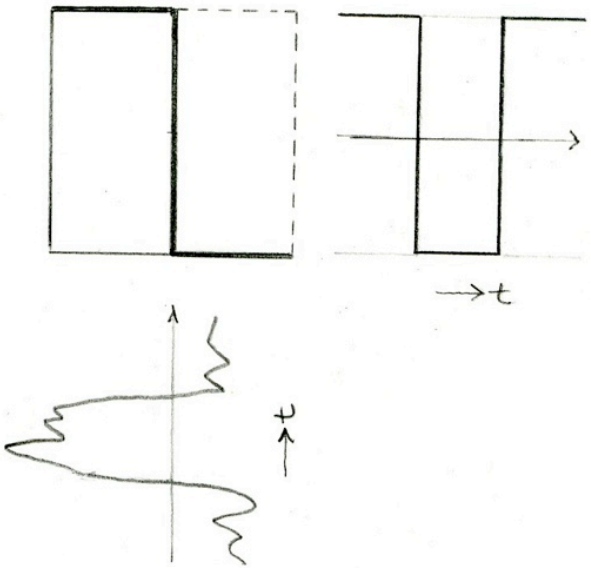
Inverter Example of Restoration

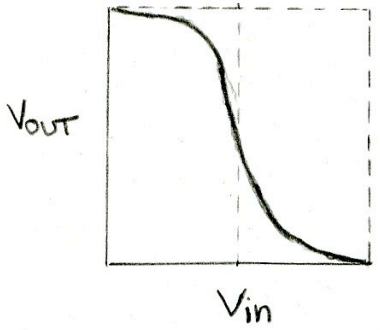
Example (look at 1-input gate, to keep it simple):



Idealize Inverter







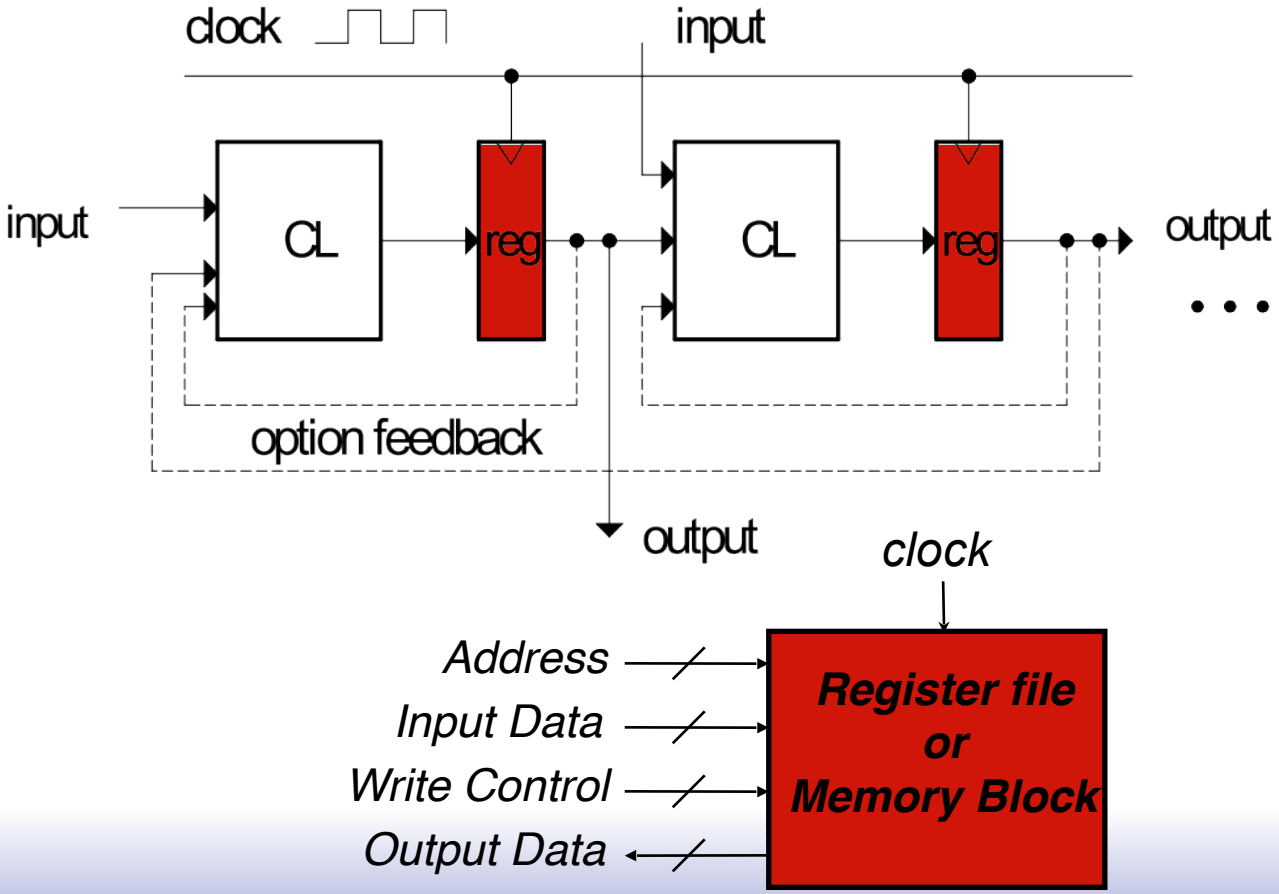
Actual Inverter

- 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.

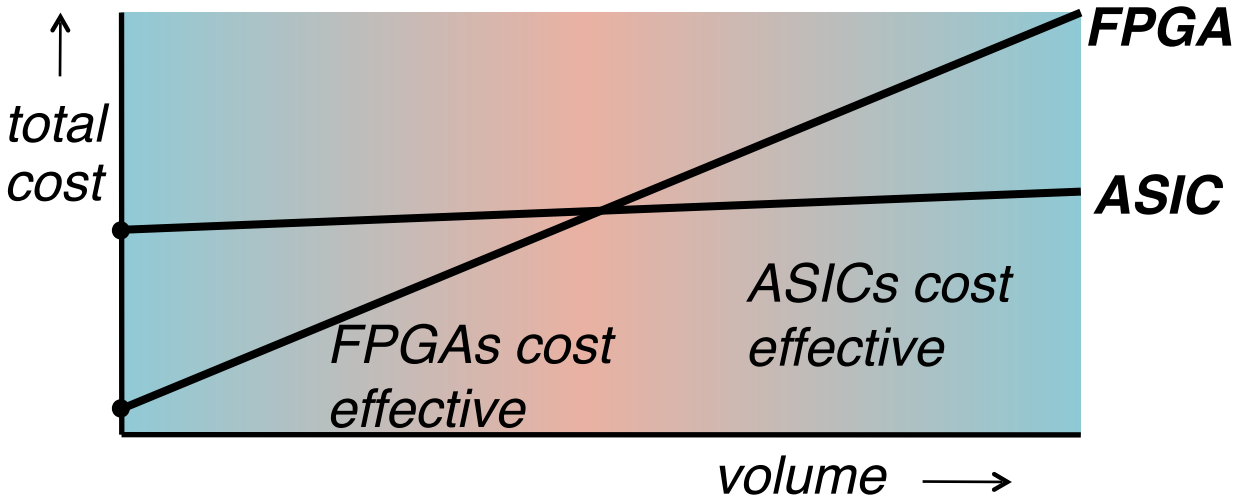
Implementation Alternative Summary

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!

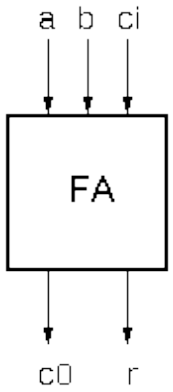
*Describing hardware with a language is similar, however, to writing a parallel program.

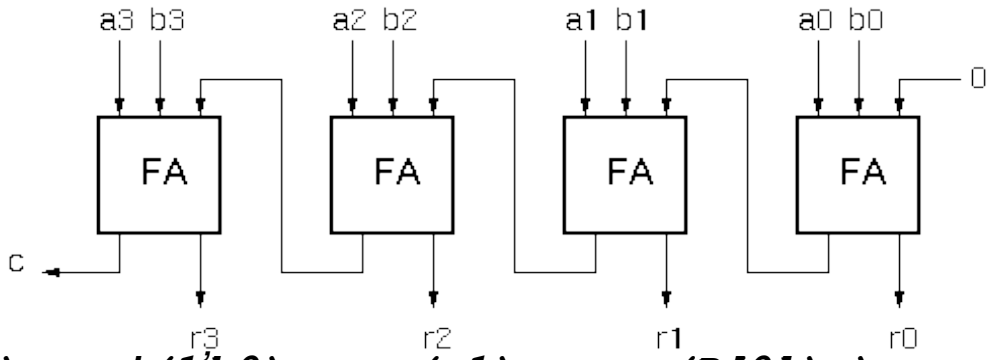
Review - Ripple Adder Example

```
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.

State Elements in Verilog

Always blocks are the only way to specify the "behavior" of state elements. Synthesis tools will turn state element behaviors into state element instances.

D-flip-flop with synchronous set and reset example:

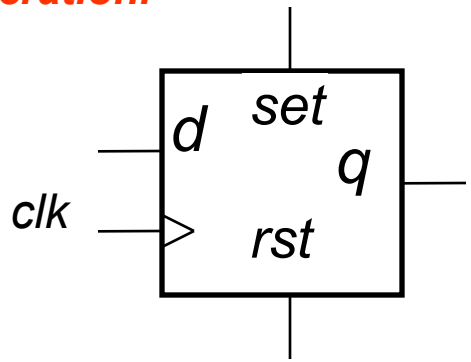
```
module dff(q, d, clk, set, rst);
  input d, clk, set, rst;
  output q;
  reg q;

  always @(posedge clk)
    if (rst)
      q <= 1'b0;
    else if (set)
      q <= 1'b1;
    else
      q <= d;
endmodule
```

keyword

"always @ (posedge clk)" is key to flip-flop generation.

This gives priority to reset over set and set over d.

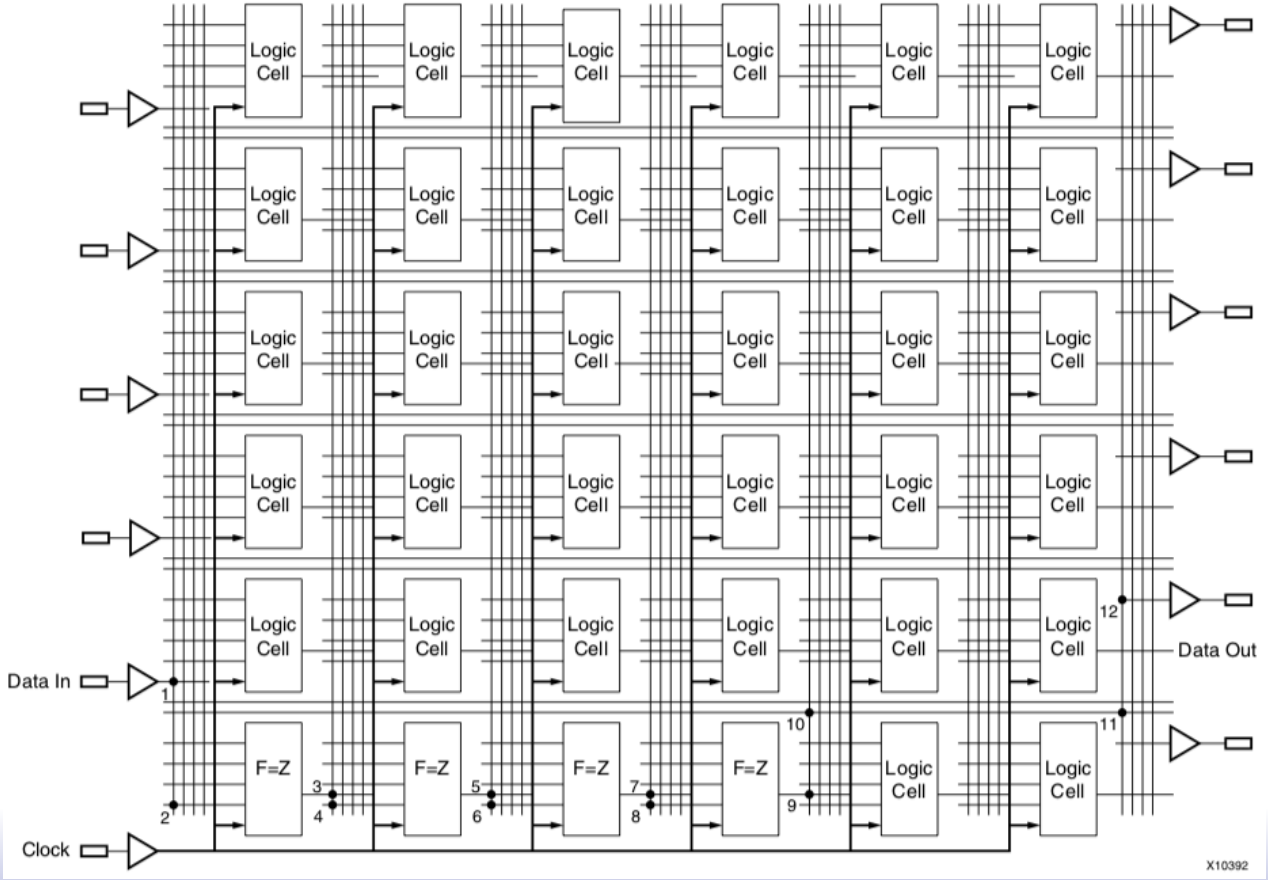


On FPGAs, maps to native flip-flop.

Unlike logic gates, their are no primitive flip-flops in Verilog. Although, it is possible to instantiate FPGA or Standard-cell specific flip-flops.

FPGA Overview

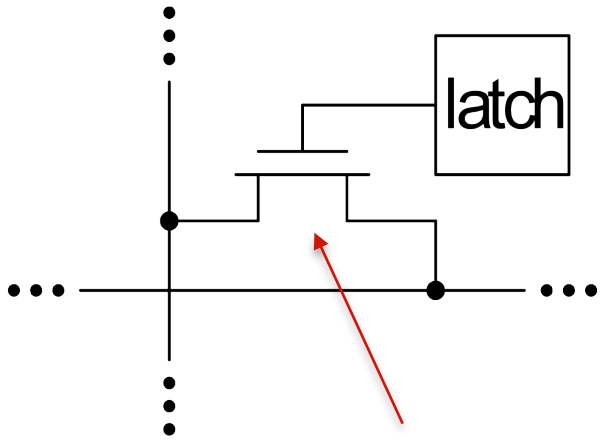
- Basic idea: two-dimensional array of logic blocks and flip-flops with a means for the user to configure (program):
  1. the interconnection between the logic blocks,
  2. the function of each block.



Simplified version of FPGA internal architecture

User Programmability

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



MOSFET used as a "switch"

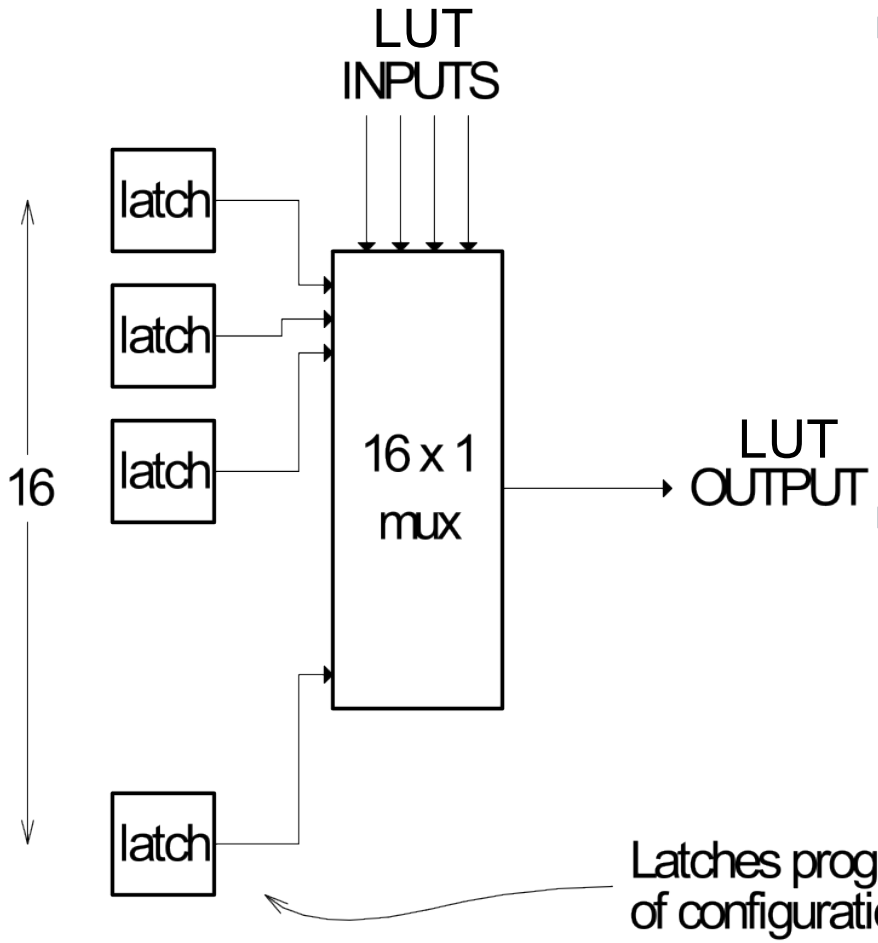
\+ reconfigurable

– volatile

– relatively large.

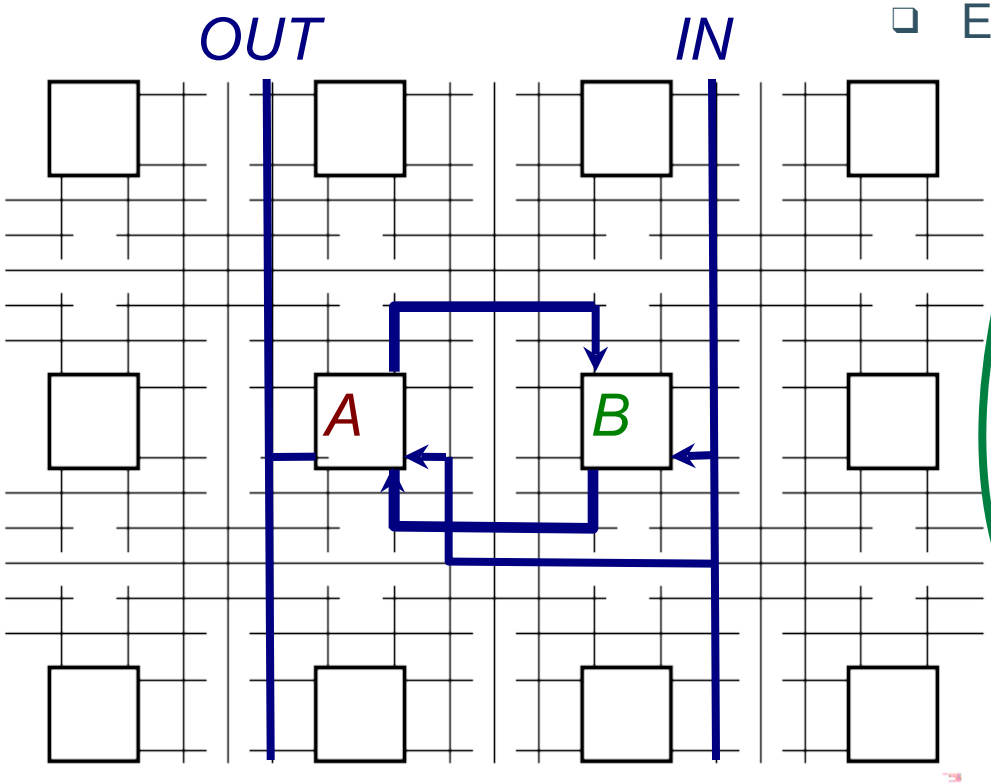
- Latches are used to:
  1. control a switch to make or break cross-point connections in the interconnect
  2. define the function of the logic blocks
  3. set user options:
     - within the logic blocks
     - in the input/output blocks
     - global reset/clock
- "Configuration bit stream" is loaded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

4-LUT Implementation

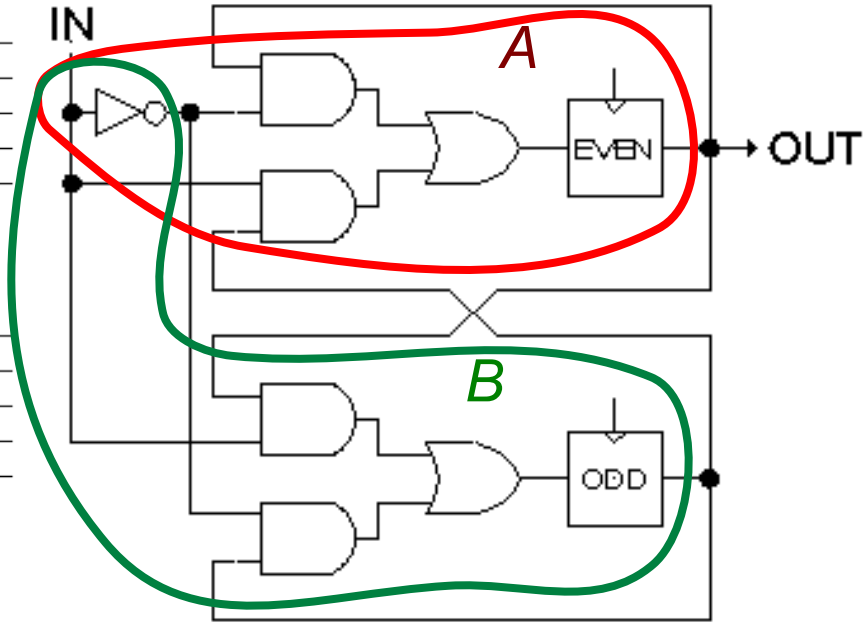


- n-bit LUT is implemented as a 2^n x 1 memory:
  - inputs choose one of 2^n memory locations.
  - memory locations (latches) are normally loaded with values from user's configuration bit stream.
  - Inputs to mux control are the CLB inputs.
- Result is a general purpose "logic gate".
  - n-LUT can implement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

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

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

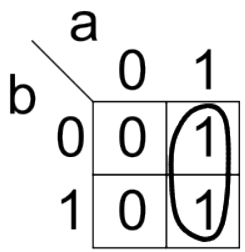
One product (and) term for each 1 in f:

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

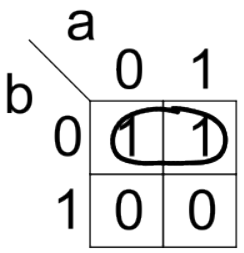
What is the cost?

Karnaugh Map Method

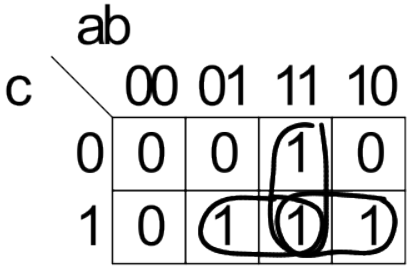
- Adjacent groups of 1's represent product terms



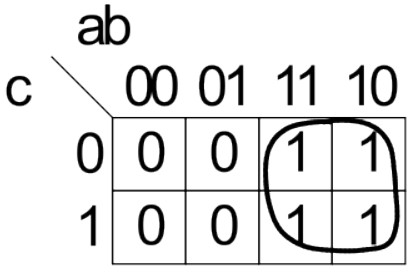
f = a



g = b'



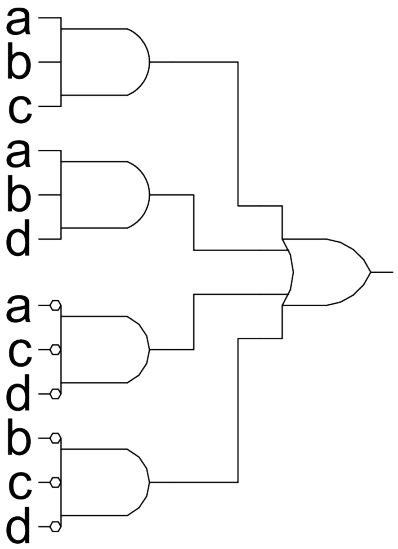
cout = ab + bc + ac



f = a

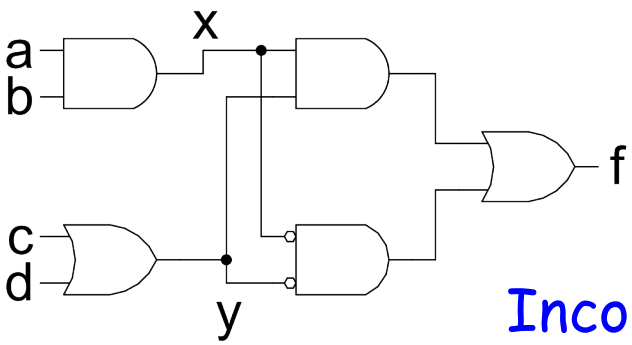
Multi-level Combinational Logic

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



let **`x = ab  y = c+d`**

`f = xy + x'y'`



Incorporates fanout.

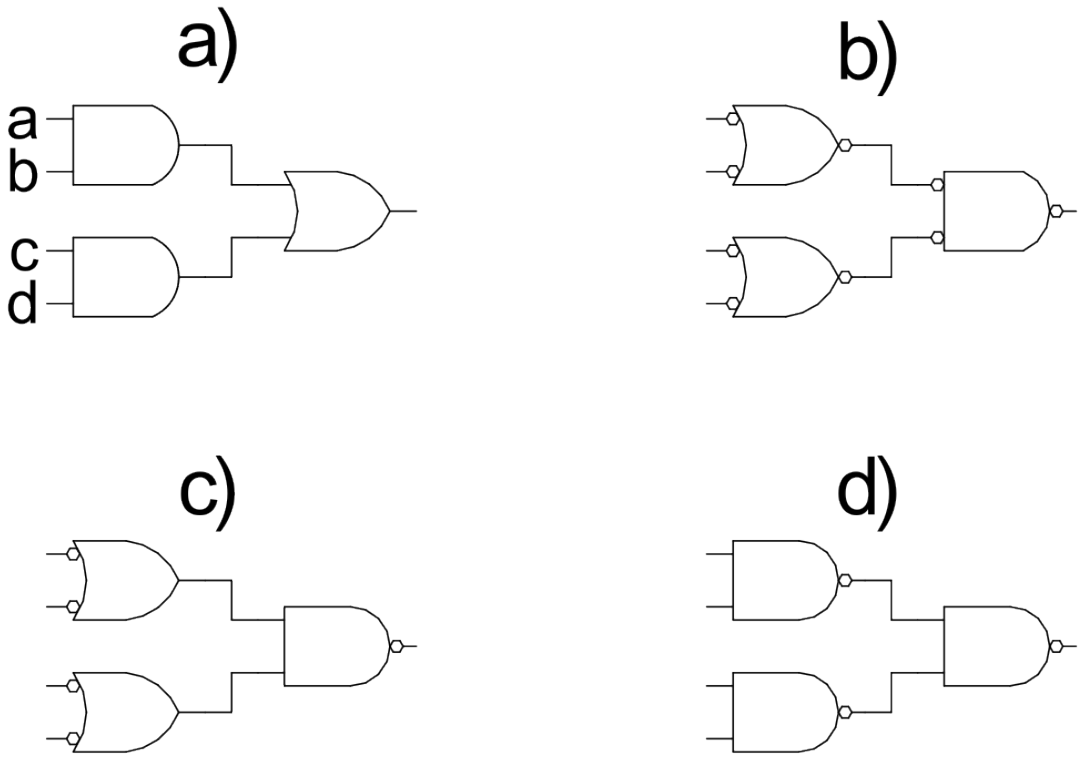
No convenient hand methods exist for multi-level logic simplification:

a) CAD Tools use sophisticated algorithms and heuristics
   Guess what? These problems tend to be NP-complete

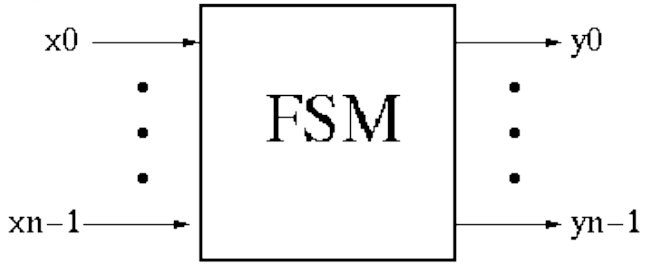
b) Humans and tools often exploit some special structure (example adder)

NAND-NAND & NOR-NOR Networks

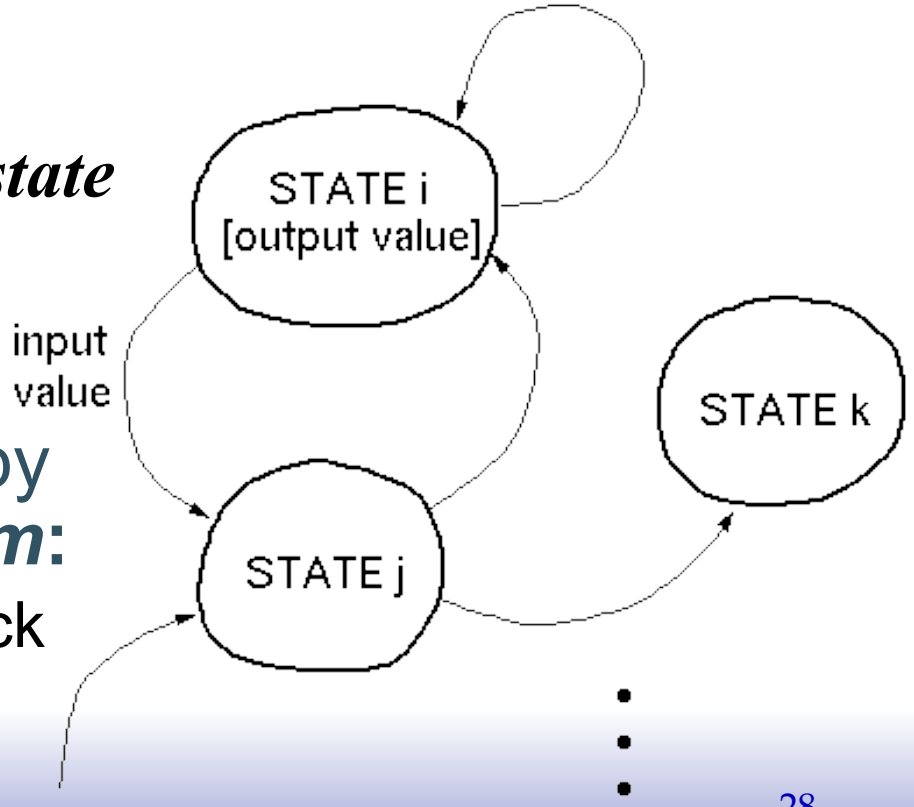
- Mapping from AND/OR to NAND/NAND



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***

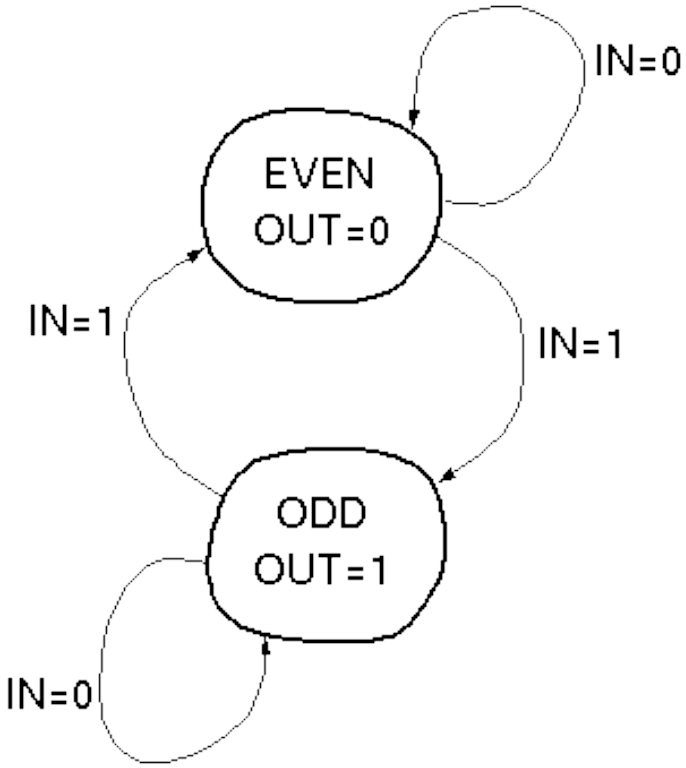


- Behavior is represented by ***State Transition Diagram***:
  - traverse one edge per clock cycle.

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

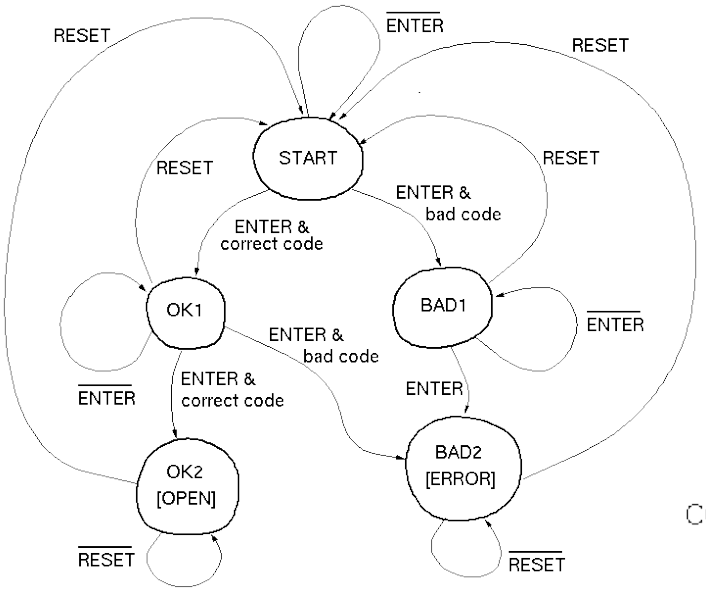
present state (ps)	OUT	IN	next state (ns)
0	0	0	0
0	0	1	1
1	1	0	1
1	1	1	0

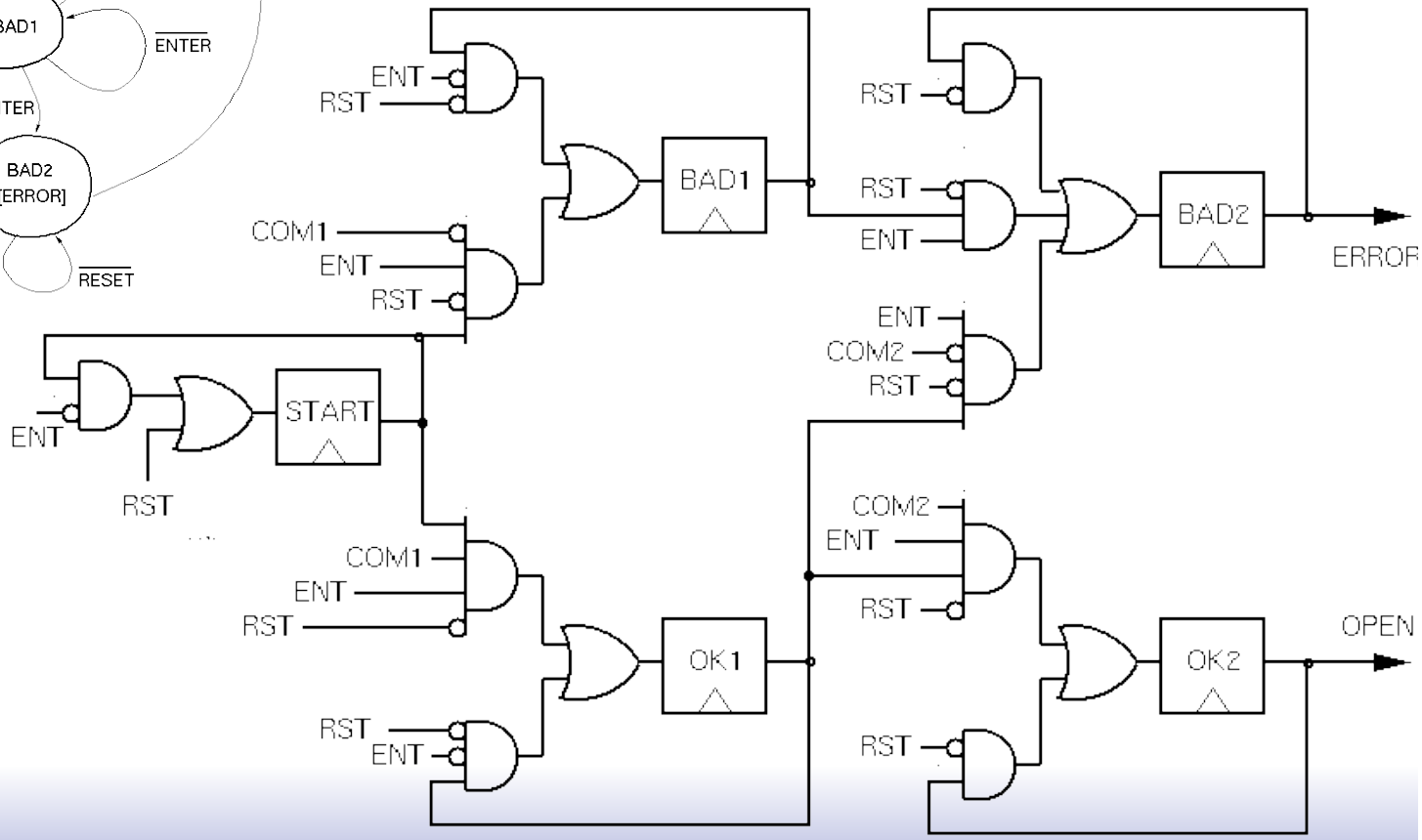
Derive logic equations from table (how?):

OUT = PS

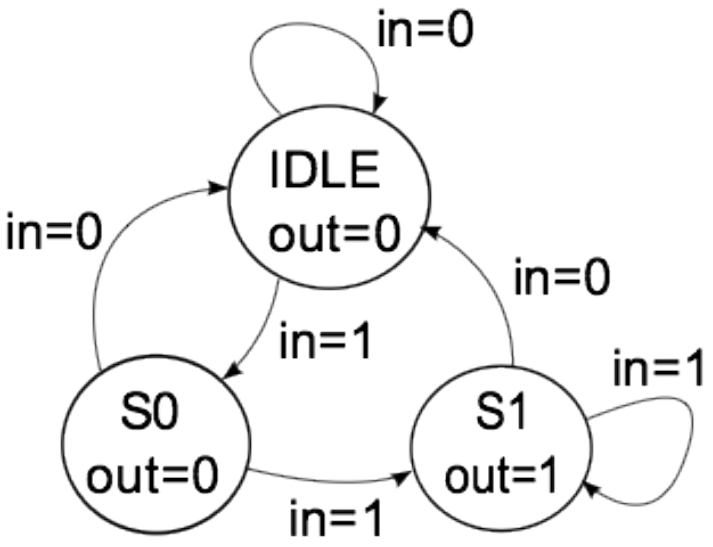
NS = PS xor IN

One-hot encoded combination lock





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*

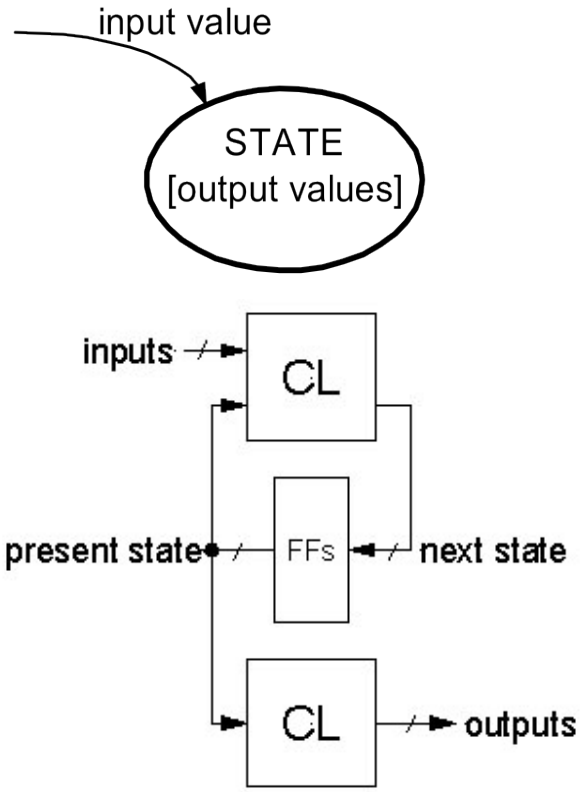
Normal values: used unless specified below.

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

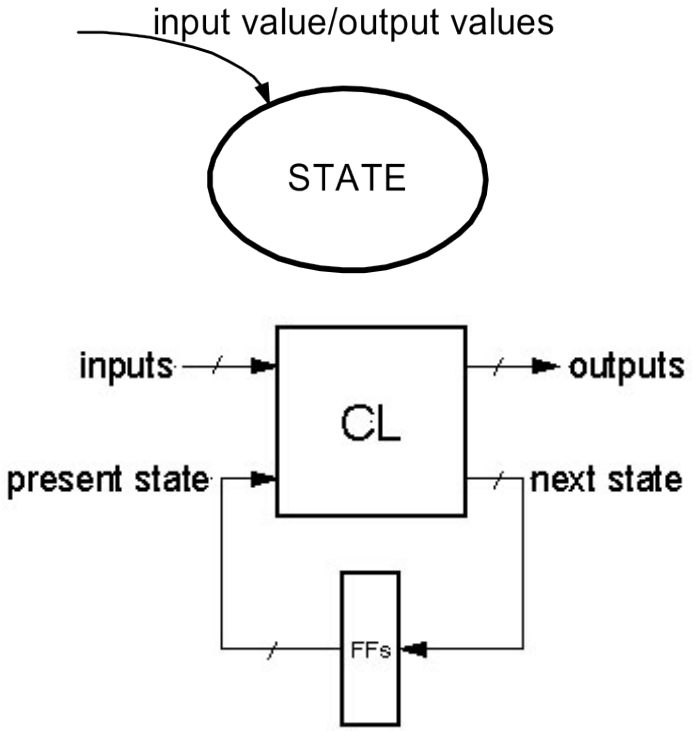
Note: The use of "blocking assignments" allow signal values to be "rewritten", simplifying the specification.

FSM Recap

Moore Machine

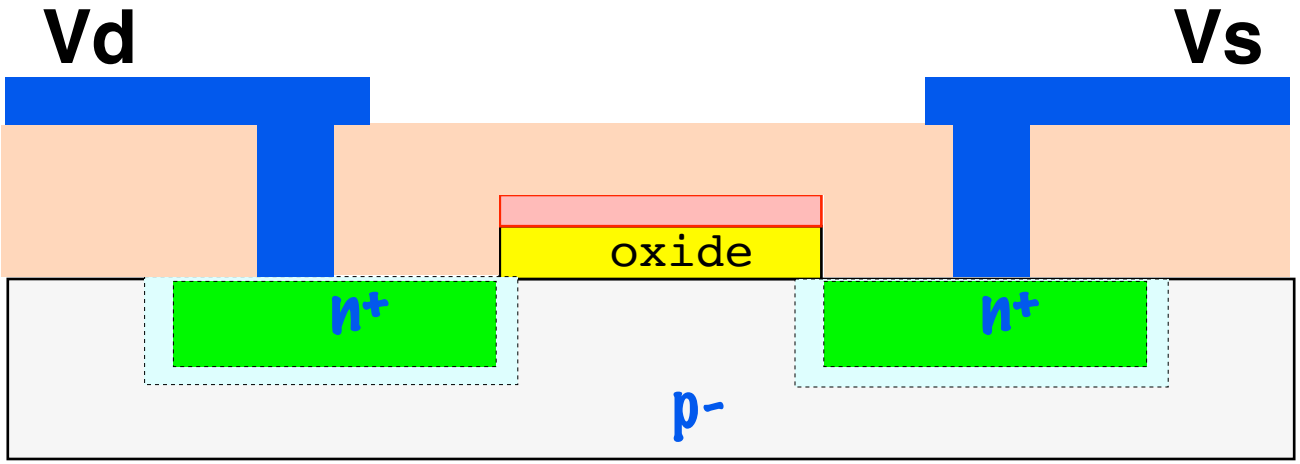


Mealy Machine



Both machine types allow one-hot implementations.

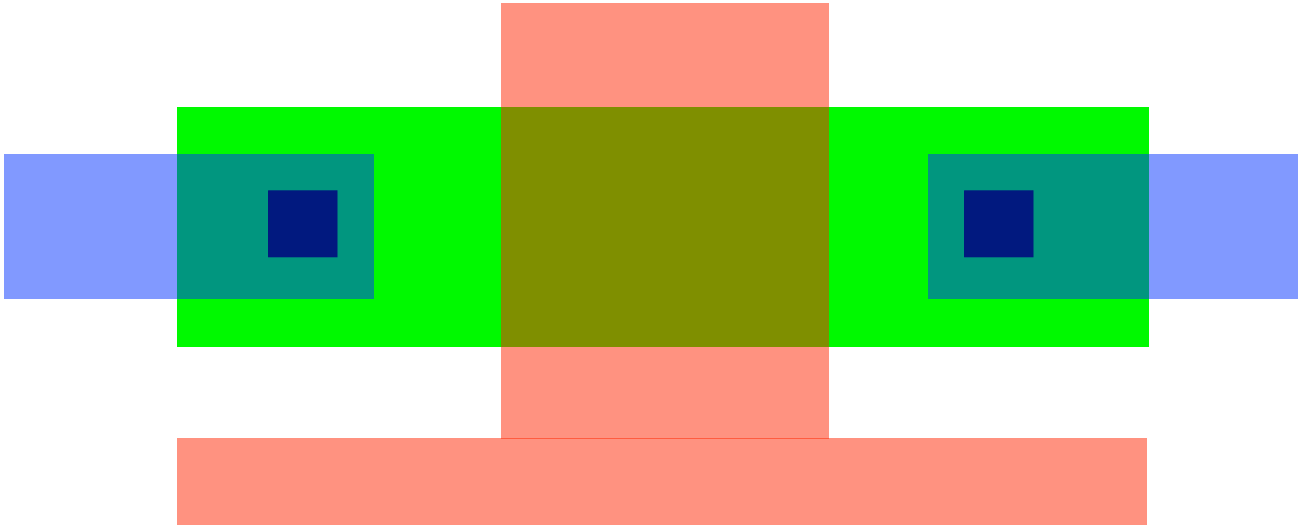
Final product ...



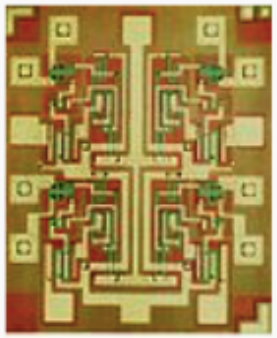
"The planar process"

Jean Hoerni, Fairchild Semiconductor 1958

Top-down view:



Semiconductor manufacturing processes



10 μm – 1971
6 μm – 1974
3 μm – 1977
1.5 μm – 1982
1 μm – 1985
800 nm – 1989
600 nm – 1994
350 nm – 1995
250 nm – 1997
180 nm – 1999
130 nm – 2001
90 nm – 2004
65 nm – 2006
45 nm – 2008
32 nm – 2010
22 nm – 2012
14 nm – 2014
10 nm – 2017
7 nm – 2018
5 nm – *~2020*

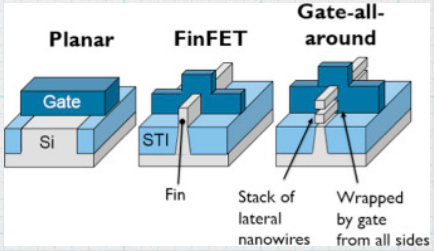
7nm

As of September 2018, mass production of 7 nm devices has begun. The first mainstream 7 nm mobile processor intended for mass market use, the Apple A12 Bionic, was released at their September 2018 event. Although Huawei announced its own 7 nm processor before the Apple A12 Bionic, the Kirin 980 on August 31, 2018, the Apple A12 Bionic was released for public, mass market use to consumers before the Kirin 980. Both chips are manufactured by TSMC. AMD is currently working on their "Rome" workstation processors, which are based on the 7 nanometer node and feature up to 64 cores.

5nm

The 5 nm node was once assumed by some experts to be the end of Moore's law. Transistors smaller than 7 nm will experience quantum tunnelling through the gate oxide layer. Due to the costs involved in development, 5 nm is predicted to take longer to reach market than the two years estimated by Moore's law. Beyond 7 nm, it was initially claimed that major technological advances would have to be made to produce chips at this small scale. In particular, it is believed that 5 nm may usher in the successor to the FinFET, such as a gate-all-around architecture.

Although Intel has not yet revealed any specific plans to manufacturers or retailers, their 2009 roadmap projected an end-user release by approximately 2020. In early 2017, Samsung announced production of a 4 nm node by 2020 as part of its revised roadmap. On January 26th 2018, TSMC announced production of a 5 nm node by 2020 on its new fab 18. In October 2018, TSMC disclosed plans to start risk production of 5 nm devices in April 2019.



3.5nm

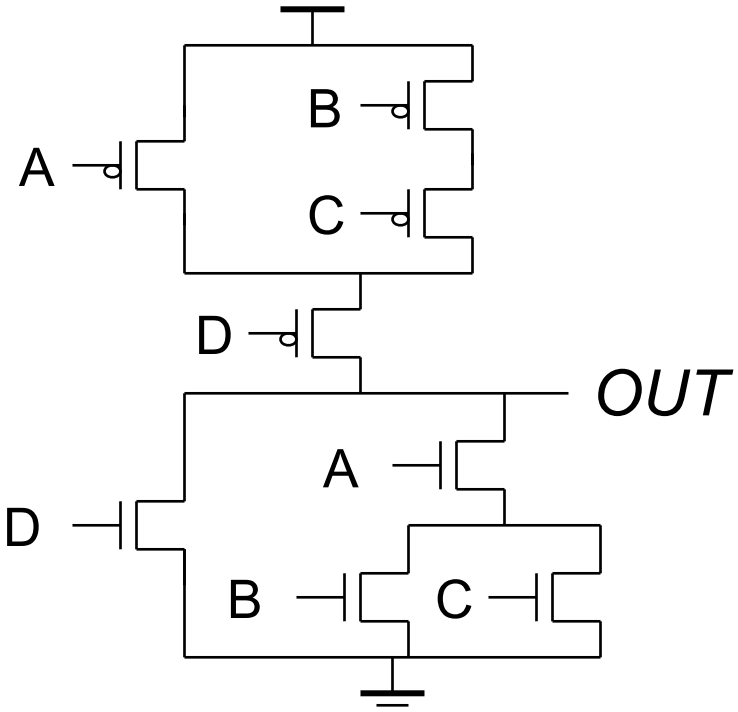
3.5 nm is a name for the first node beyond 5 nm. In 2018, IMEC and Cadence had taped out 3 nm test chips. Also, Samsung announced that they plan to use Gate-All-Around technology to produce 3 nm FETs in 2021.

* From Wikipedia

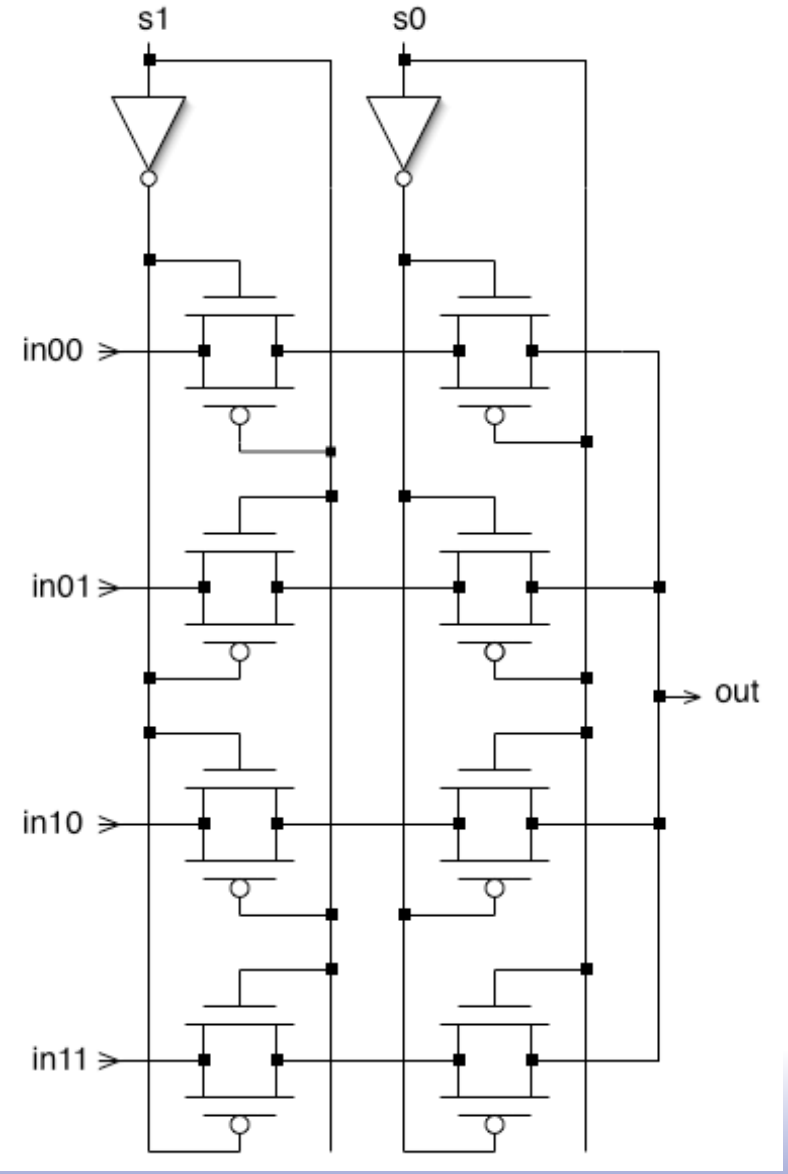
Complex CMOS Gate

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

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



4-to-1 Transmission-gate Mux

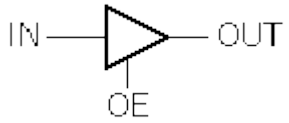


- The series connection of pass-transistors in each branch effectively forms the AND of s1 and s0 (or their complement).
- Compare cost to logic gate implementation

Any better solutions?

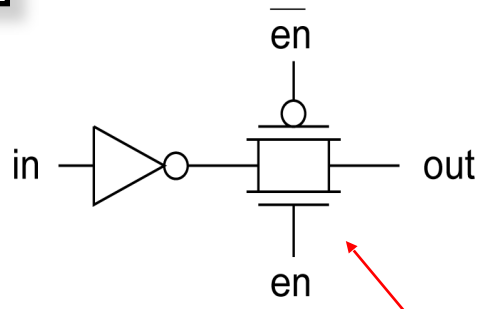
Tri-state Buffers

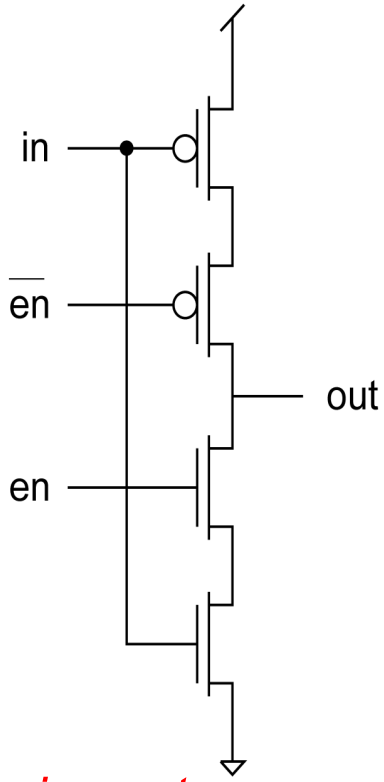
Tri-state Buffer:



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

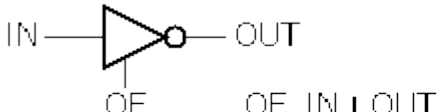
"high impedance" (output disconnected)





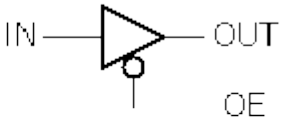
transmission gate useful in implementation

Variations:



OE	IN	OUT
0	–	Z
1	0	1
1	1	0

Inverting buffer

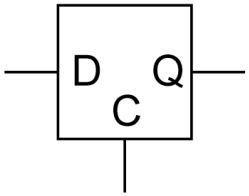


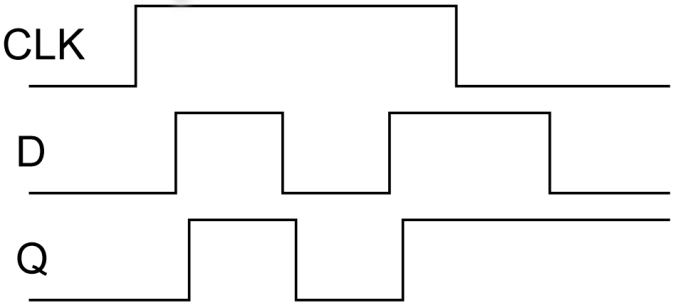
OE	IN	OUT
0	0	0
0	1	1
1	–	Z

Inverted enable

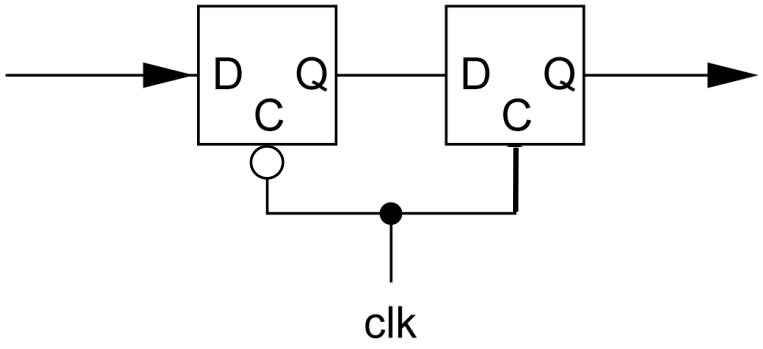
Latches and Flip-flops

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

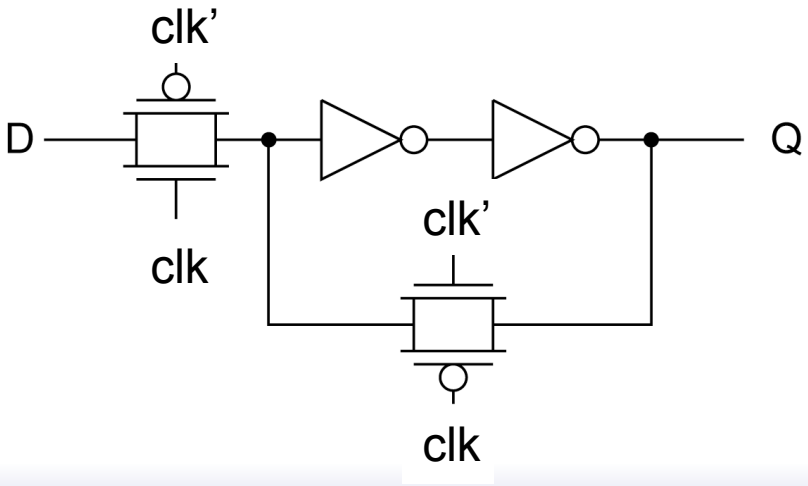




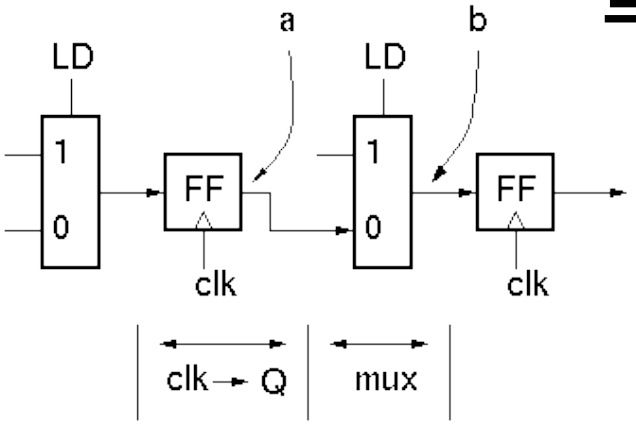
Positive Edge-triggered **flip-flop** built from two level-sensitive latches:



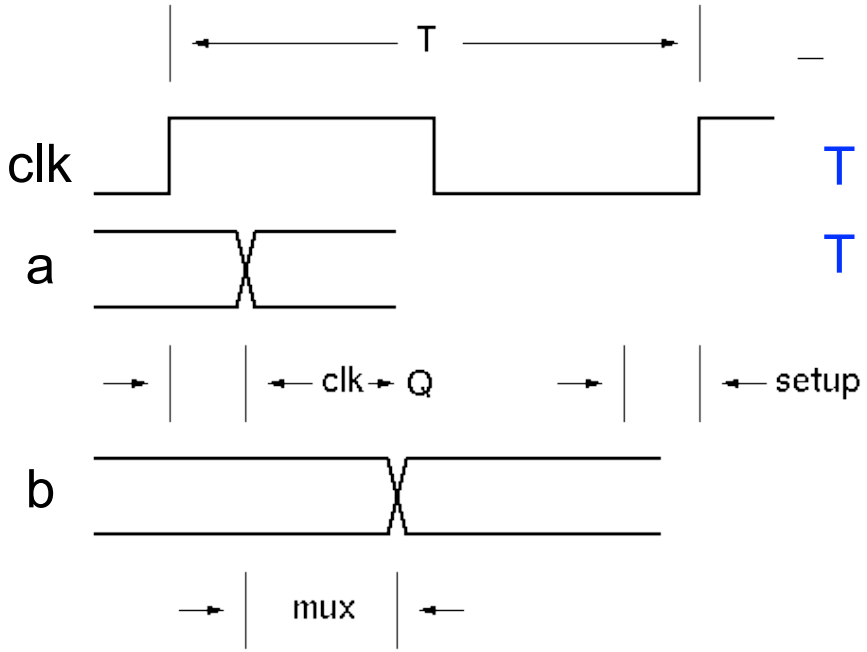
Latch Implementation:



Example



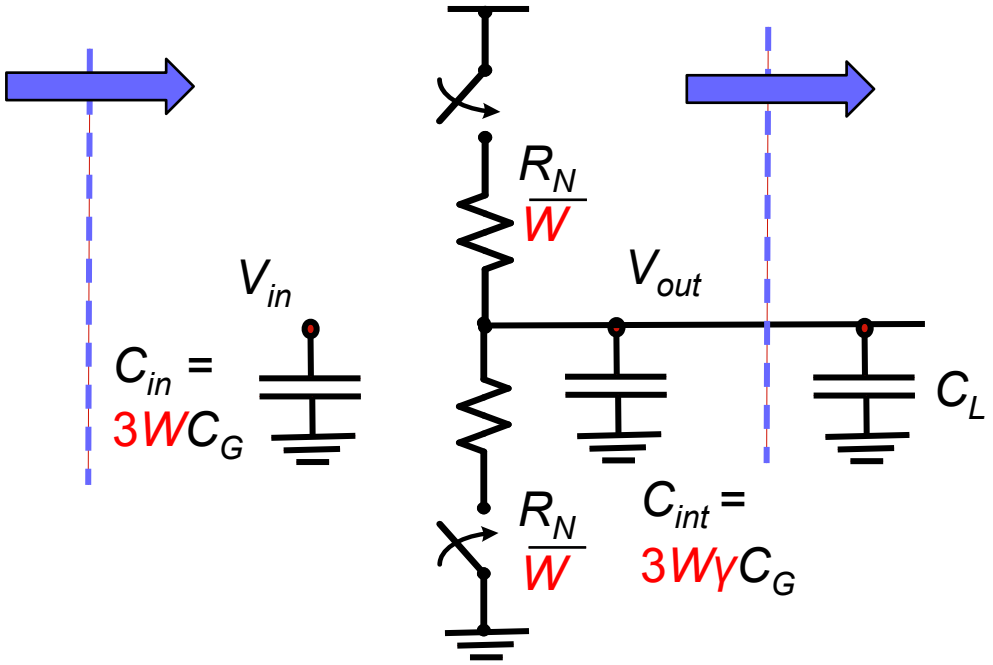
Parallel to serial converter circuit



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

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

Inverter with Load Capacitance



$$t_p = 0.69(R_N/W)(C_{int} + C_L)$$

$$= 0.69(R_N/W)(3W\gamma C_G + C_L)$$

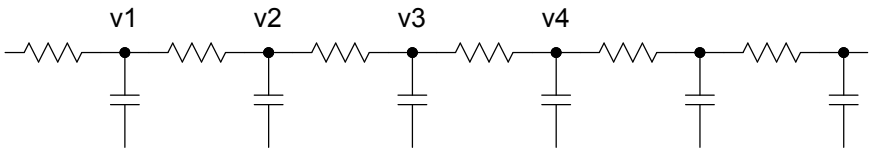
$$= 0.69(3\gamma R_N C_G)(1 + \frac{C_L}{\gamma C_{in}})$$

$$= t_{inv}(1 + \frac{C_L}{\gamma C_{in}}) = t_{p0}(1 + f/\gamma)$$

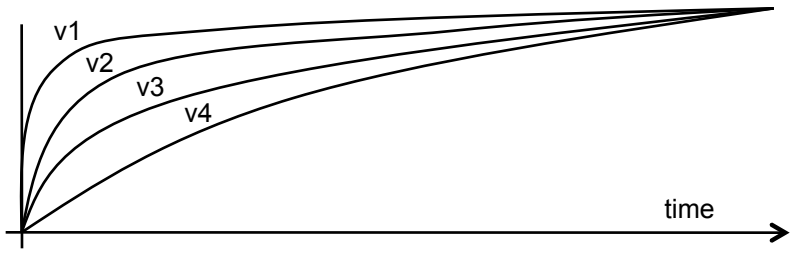
f = **fanout** = ratio between load and input capacitance of gate

Wire Delay

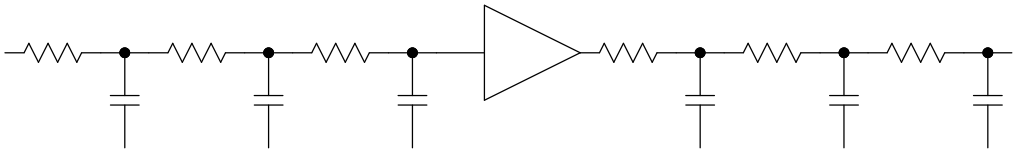
- Even in those cases where the transmission line effect is negligible:
  - Wires posses distributed resistance and capacitance



  - 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.

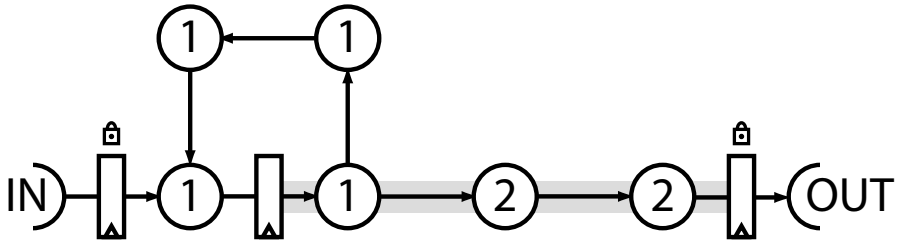


Figure 1: A small graph before retiming. The nodes represent logic delays, with the inputs and outputs passing through mandatory, fixed registers. The critical path is 5.

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

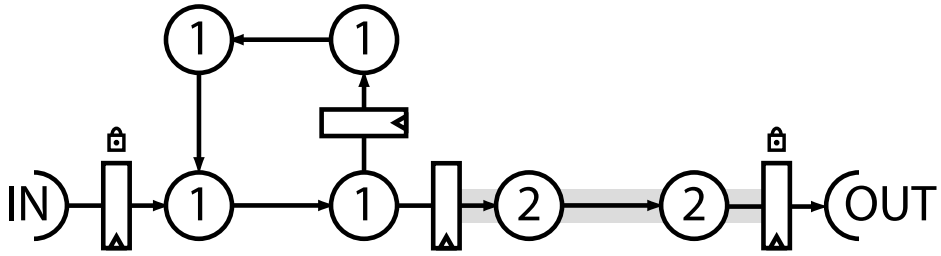
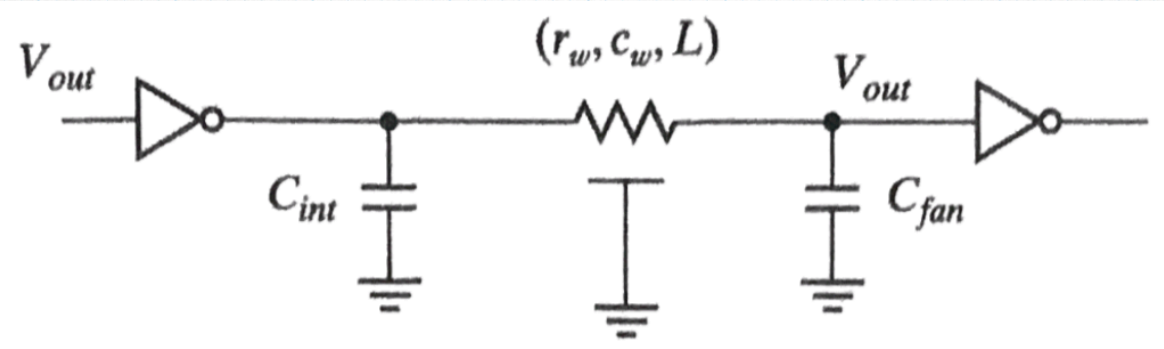


Figure 2: The example in Figure 2 after retiming. The critical path is reduced from 5 to 4.

Logic Synthesis tools can do this in simple cases.

Gate Driving long wire and other gates



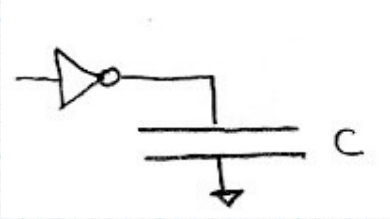
$$R_w = r_w L, \quad C_w = c_w L$$

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

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

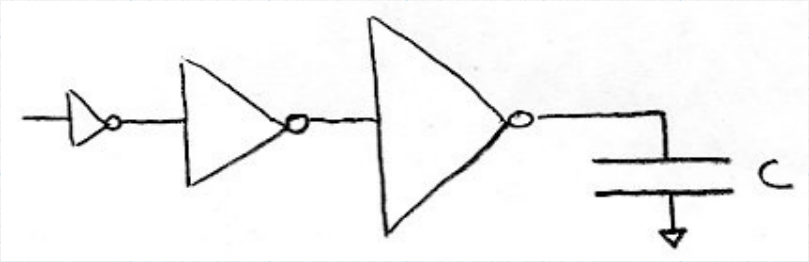
Driving Large Loads

- 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?